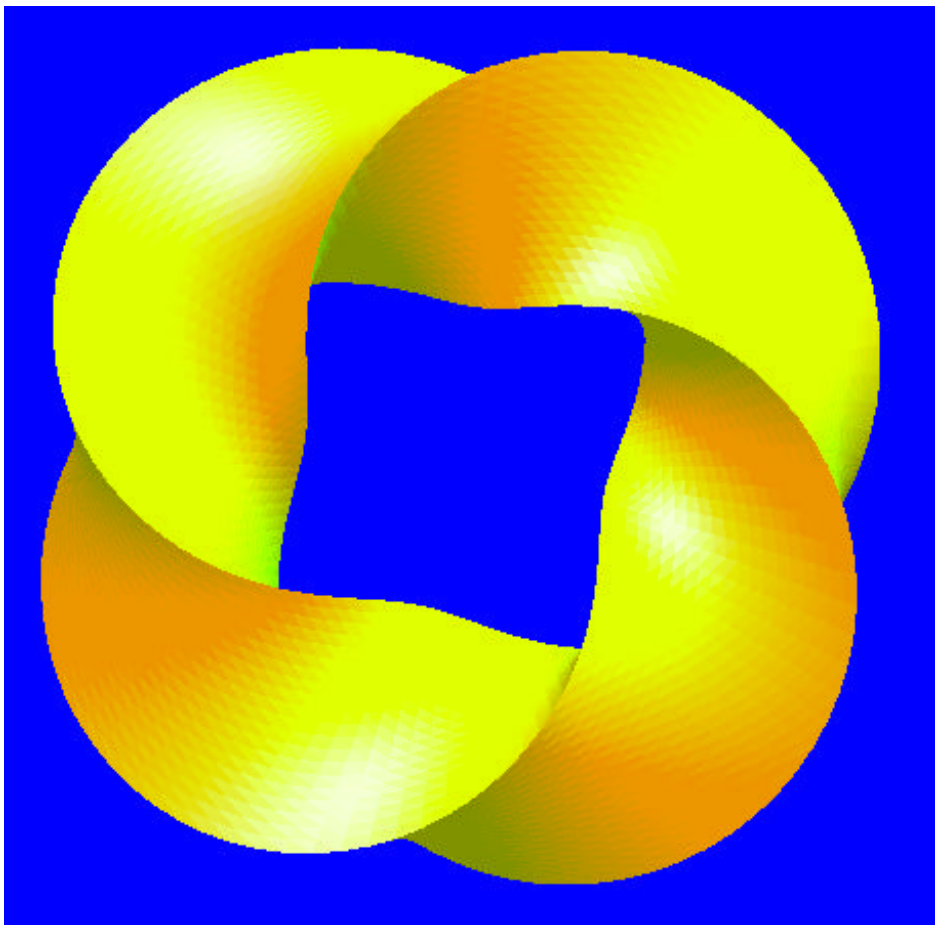
Computer Graphics: Programming, Problem Solving, and Visual Communication



Dr. Steve Cunningham
Computer Science Department
California State University Stanislaus
Turlock, CA 95382

These notes are a draft of a textbook for an introductory computer graphics course that emphasizes graphics programming and is intended for undergraduate students who have a sound background in programming. Its goal is to introduce fundamental concepts and processes for computer graphics, give students experience in computer graphics programming using the OpenGL application programming interface (API), and show the power of visual communication and of computer graphics in the sciences.

The contents below represent a relatively mature version of these notes, although some reorganization of the material is still expected and some additional topics may be developed. We hope you will find these notes to be helpful in learning computer graphics and encourage you to give us feedback with the feedback pages at the end of the notes.

CONTENTS:

- Double buffering
- Clipping planes

- Stereo viewing
- Implementation of viewing and projection in OpenGL
  - Defining a window and viewport
  - Reshaping the window
  - Defining a viewing environment
  - Defining perspective projection
  - Defining an orthogonal projection
  - Managing hidden surface viewing
  - Setting double buffering
  - Defining clipping planes
- Implementing a stereo view
- Questions
- Exercises
- Experiments

Chapter 2: Principles of Modeling

- Introduction

Simple Geometric Modeling

- Introduction
- Definitions
- Some examples
  - Point and points
  - Line segments
  - Connected lines
  - Triangle
  - Sequence of triangles
  - Quadrilateral
  - Sequence of quads
  - General polygon
  - Polyhedron
  - Aliasing and antialiasing
  - Normals
  - Data structures to hold objects
  - Additional sources of graphic objects
  - A word to the wise

Transformations and modeling

- Introduction
- Definitions
  - Transformations
  - Composite transformations
  - Transformation stacks and their manipulation
  - Compiling geometry
- A word to the wise

Scene graphs and modeling graphs

- Introduction
- A brief summary of scene graphs
  - An example of modeling with a scene graph
- The viewing transformation
- The scene graph and depth testing
- Using the modeling graph for coding
  - Example
  - Using standard objects to create more complex scenes

- Questions
- Exercises
- Experiments

Chapter 3: Implementing Modeling in OpenGL

- The OpenGL model for specifying geometry
  - Point and points mode
  - Line segments
  - Line strips
  - Line loops
  - Triangle
  - Sequence of triangles
  - Quads
  - Quad strips
  - General polygon
  - Antialiasing
  - The cube we will use in many examples
- Additional objects with the OpenGL toolkits
  - GLU quadric objects
    - > GLU cylinder
    - > GLU disk
    - > GLU sphere
  - The GLUT objects
  - An example
- A word to the wise
- Transformations in OpenGL
- Code examples for transformations
  - Simple transformations
  - Transformation stacks
  - Inverting the eyepoint transformation
  - Creating display lists

Chapter 4: Mathematics for Modeling

- Coordinate systems and points
- Points, lines, and line segments
- Distance from a point to a line
- Line segments and parametric curves
- Vectors
- Dot and cross products of vectors
- Reflection vectors
- Transformations
- Planes and half-spaces
- Distance from a point to a plane
- Polygons and convexity
- Polyhedra
- Collision detection
- Polar, cylindrical, and spherical coordinates
- Higher dimensions?

Chapter 5: Color and Blending

- Introduction
- Definitions
  - The RGB cube
  - Luminance

- Other color models
- Color depth
- Color gamut
- Color blending with the alpha channel
- Challenges in blending
- Modeling transparency with blending
- Indexed color
- Using color to create 3D images

• Some examples
- An object with partially transparent faces

• Color in OpenGL
- Enabling blending

• A word to the wise
• Code examples
- A model with parts having a full spectrum of colors
- The HSV cone
- The HLS double cone
- An object with partially transparent faces

• Questions
• Exercises
• Experiments

Chapter 6: Visual Communication

• Introduction
• General issues in visual communication
- Use appropriate representation for your information
- Keep your images focused
- Use appropriate presentation levels for your information
- Use appropriate forms for your information
- Be very careful to be accurate with your information
- Understand and respect the cultural context of your audience
- Make your interactions reflect familiar and comfortable relationships between cause and effect

• Shape
- Comparing shape and color encodings

• Color
- Emphasis colors
- Background colors
- Color deficiencies in audiencd
- Naturalistic color
- Pseudocolor and color ramps
- Implementing color ramps
- Using color ramps
- To light or not to light
- Higher dimensions

• Dimensions
• Image context
- Choosing an appropriate view
- Legends to help communicate your encodings
- Labels to help communicate your problem

• Motion
- Leaving traces of motion
- Motion blurring

• Interactions

- Cultural context of the audience
- Accuracy
- Output media
- Implementing some of these ideas in OpenGL
  - Using color ramps
  - Legends and labels
  - Creating traces
  - Using the accumulation buffer
- A word to the wise

Chapter 7: Graphical Problem Solving in Science

- Introduction
- Examples
- Diffusion
  - Temperatures in a bar
  - Spread of disease
- Function graphing and applications
- Parametric curves and surfaces
- Graphical objects that are the results of limit processes
- Scalar fields
- Representation of objects and behaviors
  - Gas laws and diffusion principles
  - Molecular display
  - Monte Carlo modeling process
- 4D graphing
  - Volume data
  - Vector fields
- Graphing in higher dimensions
- Data-driven graphics
- Code examples
  - Diffusion
  - Function graphing
  - Parametric curves and surfaces
  - Limit processes
  - Scalar fields
  - Representation of objects and behaviors
  - Molecular display
  - Monte Carlo modeling
  - 4D graphing
  - Higher dimensional graphing
  - Data-driven graphics
- Credits

Chapter 8: The Rendering Pipeline

- Introduction
- The pipeline
- The rendering pipeline for OpenGL
  - Texture mapping in the rendering pipeline
  - Per-fragment operations
  - Some extensions to OpenGL
  - An implementation of the rendering pipeline in a graphics card
- The rasterization process

Chapter 9: Lighting and Shading

Lighting

- Definitions
  - Ambient, diffuse, and specular light
  - Surface normals
- Light properties
  - Positional lights
  - Spotlights
  - Attenuation
  - Directional lights
  - Positional and moving lights

Materials

Shading

- Definitions
  - Flat shading
  - Smooth shading
- Examples of flat and smooth shading
- Calculating per-vertex normals
  - Averaging polygon normals
  - Analytic computations
- Other shading models
  - Vertex and pixel shaders

Global Illumination

Local Illumination

- Lights and materials in OpenGL
  - Specifying and defining lights
  - Defining materials
  - Setting up a scene to use lighting
  - Using GLU quadric objects
  - An example: lights of all three primary colors applied to a white surface
  - Code for the example
- A word to the wise
  - Shading example
- Questions
- Exercises
- Experiments

Chapter 10: Event Handling

- Introduction
- Definitions
- Some examples of events
  - keypress events
  - mouse events
  - menu events
  - window events
  - system events
  - software events
- The vocabulary of interaction
- A word to the wise
- Events in OpenGL
- Callback registering
- Some details
- Code examples
  - Idle event callback

- Keyboard callback
- Menu callback
- Mouse callback for object selection
- Mouse callback for mouse motion

The MUI (Micro User Interface) Facility

- Introduction
- Definitions
  - Menu bars
  - Buttons
  - Radio buttons
  - Text boxes
  - Horizontal sliders
  - Vertical sliders
  - Text labels
- Using the MUI functionality
- Some examples
- Installing MUI for Windows
- A word to the wise

Chapter 11: Texture Mapping

- Introduction
- Definitions
  - 1D texture maps
  - 2D texture maps
  - 3D texture maps
  - Associating a vertex with a texture point
  - The relation between the color of the object and the color of the texture map
  - Other meanings for texture maps
- Creating a texture map
  - Getting an image as a texture map
  - Generating a synthetic texture map
- Texture mapping and billboards
- Interpolation for texture maps
- Antialiasing in texturing
- MIP mapping
- Multitexturing
- Using billboards
- Texture mapping in OpenGL
  - Associating vertices and texture points
  - Capturing a texture from the screen
  - Texture environment
  - Texture parameters
  - Getting and defining a texture map
  - Texture coordinate control
  - Texture interpolation
  - Texture mapping and GLU quadrics
- Some examples
  - The Chromadepth™ process
  - Using 2D texture maps to add interest to a surface
  - Environment maps
- A word to the wise
- Code examples
  - A 1D color ramp
  - An image on a surface

    - An environment map
    - Multitexturing code
- Questions
- Exercises
- Experiments

Chapter 12: Dynamics and Animation

- Introduction
- Definitions
- Keyframe animation
    - Temporal aliasing
    - Building an animation
- Some examples
    - Moving objects in your model
    - Moving parts of objects in your model
    - Moving the eye point or the view frame in your model
    - Changing features of your models
- Some points to consider when doing animations with OpenGL
- Code examples
- A word to the wise

Chapter 13: High-Performance Graphics Techniques

- Definitions
- Techniques
    - Hardware avoidance
    - Designing out visible polygons
    - Culling polygons
    - Avoiding depth comparisons
    - Front-to-back drawing
    - Binary space partitioning
    - Clever use of textures
    - System speedups
    - Level of detail
    - Reducing lighting computation
    - Fog
    - Collision detection
- A word to the wise

Chapter 14: Object Selection

- Introduction
- Picking in OpenGL
- Definitions
- Making picking work
- The pick matrix
- Using the back color buffer to do picking
- A selection example
- A word to the wise
- Questions
- Exercises
- Experiments

Chapter 15: Interpolation and Spline Modeling

- Introduction
    - Interpolations

- Extending interpolations to more control points
- Interpolations in OpenGL
  - Automatic normal and texture generation with evaluators
  - Additional techniques
- Definitions
- Some examples
  - Spline curves
  - Spline surfaces
- A word to the wise

Chapter 16: Per-Pixel Operations

- Introduction
- Definitions
- Ray casting
- Ray tracing
- Ray tracing and global illumination
- Volume rendering
- Fractal images
- Iterated function systems
- Per-pixel operations supported by OpenGL

Chapter 17: Hardcopy

- Introduction
- Definitions
  - Digital images
  - Print
  - Film
  - Video
  - Digital video
  - 3D object prototyping
  - The STL file
- A word to the wise

Appendices

- Appendix I: PDB file format
- Appendix II: CTL file format
- Appendix III: STL file format

References and Resources

- References
- Resources

Evaluation

- Instructor's evaluation
- Student's evaluation

Because this is a draft of a textbook for an introductory, API-based computer graphics course, the author recognizes that there may be some inaccuracies, incompleteness, or clumsiness in the presentation and apologizes for these in advance. Further development of these materials, as well as source code for many projects and additional examples, is ongoing continuously. All such materials will be posted as they are ready on the author's Web site:

`http://www.cs.csustan.edu/~rsc/NSF/`

Your comments and suggestions will be very helpful in making these materials as useful as possible and are solicited; please contact

Steve Cunningham
California State University Stanislaus
`rsc@cs.csustan.edu`

This work was supported by National Science Foundation grant DUE-9950121. All opinions, findings, conclusions, and recommendations in this work are those of the author and do not necessarily reflect the views of the National Science Foundation. The author also gratefully acknowledges sabbatical support from California State University Stanislaus and thanks the San Diego Supercomputer Center, most particularly Dr. Michael J. Bailey, for hosting this work and for providing significant assistance with both visualization and science content. Ken Brown, a student of the author's, provided invaluable and much-appreciated assistance with several figures and concepts in this manuscript. The author also thanks students Ben Eadington, Jordan Maynard, and Virginia Muncy for their contributions through examples, and a number of others for valuable conversations and suggestions on these notes.

Chapter 0: Getting Started

This book is intended for a beginning course in computer graphics for students with a sound programming background but no previous computer graphics experience. It includes a few features that are not found in most beginning courses:

- The focus is on computer graphics programming with a graphics API, and in particular discusses the OpenGL API. Many of the fundamental algorithms and techniques that are at the root of computer graphics are covered only at the level they are needed to understand questions of graphics programming. This differs from most computer graphics textbooks that place a great deal of emphasis on understanding these algorithms and techniques. We recognize the importance of these for persons who want to develop a deep knowledge of the subject and suggest that a second graphics course can provide that knowledge. We believe that the experience provided by API-based graphics programming will help you understand the importance of these algorithms and techniques as they are developed and will equip you to work with them more fluently than if you met them with no previous background.
- We focus on 3D graphics to the almost complete exclusion of 2D techniques. It has been traditional to start with 2D graphics and move up to 3D because some of the algorithms and techniques have been easier to grasp at the 2D level, but without that concern it seems easier simply to start with 3D and discuss 2D as a special case.
- Because we focus on graphics programming rather than algorithms and techniques, we have fewer instances of data structures and other computer science techniques. This means that these notes can be used for a computer graphics course that can be taken earlier in a student's computer science studies than the traditional graphics course. Our basic premise is that this course should be quite accessible to a student with a sound background in programming a sequential imperative language, particularly C.
- These notes include an emphasis on the scene graph as a fundamental tool in organizing the modeling needed to create a graphics scene. The concept of scene graph allows the student to design the transformations, geometry, and appearance of a number of complex components in a way that they can be implemented quite readily in code, even if the graphics API itself does not support the scene graph directly. This is particularly important for hierarchical modeling, but it provides a unified design approach to modeling and has some very useful applications for placing the eye point in the scene and for managing motion and animation.
- These notes include an emphasis on visual communication and interaction through computer graphics that is usually missing from textbooks, though we expect that most instructors include this somehow in their courses. We believe that a systematic discussion of this subject will help prepare students for more effective use of computer graphics in their future professional lives, whether this is in technical areas in computing or is in areas where there are significant applications of computer graphics.
- Many, if not most, of the examples in these notes are taken from sources in the sciences, and they include two chapters on scientific and mathematical applications of computer graphics. This makes the notes useable for courses that include science students as well as making graphics students aware of the breadth of areas in the sciences where graphics can be used.

This set of emphases makes these notes appropriate for courses in computer science programs that want to develop ties with other programs on campus, particularly programs that want to provide science students with a background that will support development of computational science or scientific visualization work.

What is a graphics API?

The short answer is that an API is an *Application Programming Interface*—a set of tools that allow a programmer to work in an application area. Thus a **graphics** API is a set of tools that allow a programmer to write applications that use computer graphics. These materials are intended to introduce you to the OpenGL graphics API and to give you a number of examples that will help

you understand the capabilities that OpenGL provides and will allow you to learn how to integrate graphics programming into your other work.

Overview of the book

The book is organized along fairly traditional lines, treating projection, viewing, modeling, rendering, lighting, shading, and many other aspects of the field, emphasizing 3D graphics and interactive techniques. It also includes an emphasis on using computer graphics to address real problems and to communicate results effectively to the viewer. As we move through this material, we describe some general principles in computer graphics and show how the OpenGL API provides the graphics programming tools that implement these principles. We do not spend time describing in depth the algorithms behind the techniques or the way the techniques are implemented; your instructor will provide these if he or she finds it necessary. Instead, the book focuses on describing the concepts behind the graphics and on using a graphics API (application programming interface) to carry out graphics operations and create images.

The book will give beginning computer graphics students a good introduction to the range of functionality available in a modern computer graphics API. They are based on the OpenGL API, but we have organized the general outline so that they could be adapted to fit another API as these are developed.

The key concept in the book, and in the computer graphics programming course, is the use of computer graphics to communicate information to an audience. We usually assume that the information under discussion comes from the sciences, and include a significant amount of material on models in the sciences and how they can be presented visually through computer graphics. It is tempting to use the word "visualization" somewhere in the title of this document, but we would reserve that word for material that is fully focused on the science with only a sidelight on the graphics; because we reverse that emphasis, the role of visualization is in the application of the graphics.

We have tried to match the sequence of chapters to the sequence we would expect to be used in an introductory course, and in some cases, the presentation of one module will depend on your knowing the content of an earlier chapter. However, in other cases it will not be critical that earlier chapters have been covered. It should be pretty obvious if other chapters are assumed, and we may make that assumption explicit in some modules.

What is Computer Graphics?

We view computer graphics as the art and science of creating synthetic images by programming the geometry and appearance of the contents of the images, and by displaying the results of that programming on appropriate display devices that support graphical output. The programming may be done (and in these notes, is assumed to be done) with the support of a graphics API that does most of the detailed work of rendering the scene that the programming defines.

The work of the programmer is to develop representations for the geometric entities that are to make up the images, to assemble these entities into an appropriate geometric space where they can have the proper relationships with each other as needed for the image, to define and present the look of each of the entities as part of that scene, to specify how the scene is to be viewed, and to specify how the scene as viewed is to be displayed on the graphic device. These processes are supported by the 3D graphics pipeline, as described below, which will be one of our primary tools in understanding how graphics processes work.

In addition to the work mentioned so far, there are two other important parts of the task for the programmer. Because a static image does not present as much information as a moving image, the

programmer may want to design some motion into the scene, that is, may want to define some animation for the image. And because a user may want to have the opportunity to control the nature of the image or the way the image is seen, the programmer may want to design ways for the user to interact with the scene as it is presented.

All of these topics will be covered as later chapters develop these ideas, using the OpenGL graphics API as the basis for implementing the actual graphics programming.

The 3D Graphics Pipeline

The 3D computer graphics pipeline is simply a process for converting coordinates from what is most convenient for the application programmer into what is most convenient for the display hardware. We will explore the details of the steps for the pipeline in the chapters below, but here we outline the pipeline to help you understand how it operates. The pipeline is diagrammed in Figure 0.1, and we will start to sketch the various stages in the pipeline here, with more detail given in subsequent chapters.

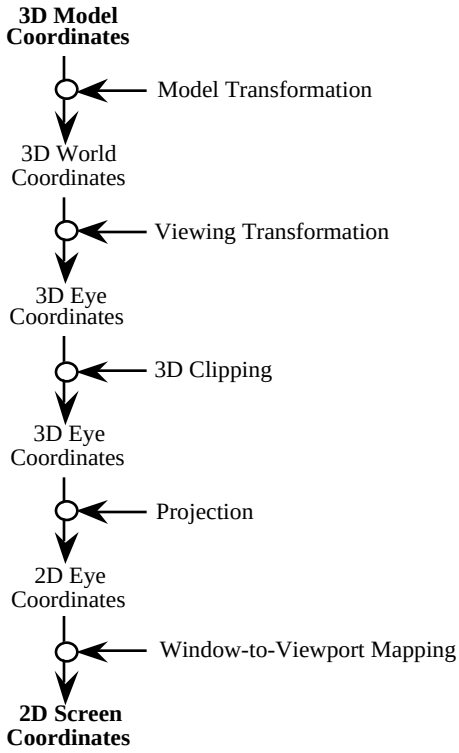


Figure 0.1: The graphics pipeline's stages and mappings

3D model coordinate systems

The application programmer starts by defining a particular object about a local origin that lies somewhere in or around the object. This is what would naturally happen if the object was created with some sort of modeling or computer-aided design system or was defined by a mathematical function. Modeling something about its local origin involves defining it in terms of *model coordinates*, a coordinate system that is used specifically to define a particular graphical object. Because the coordinate system is part of an object's design, it may be different for every part of a scene. In order to integrate each object, built with its own coordinates, into a single overall 3D

world space, the object must be placed in the world space by using an appropriate *modeling transformation*.

Modeling transformations, like all the transformations we will describe throughout the book, are functions that move objects while preserving their geometric properties. The transformations that are available to us in a graphics system are rotations, translations, and scaling. Rotations hold a line through the origin of a coordinate system fixed and rotate all the points in a scene by a fixed angle around the line, translations add a fixed value to each of the coordinates of each point in a scene, and scaling multiplies each coordinate of a point by a fixed value. These will be discussed in much more detail in the chapter on modeling below. All transformations may be represented as matrices, so sometimes in a graphics API you will see a mention of a matrix; this almost always means that a transformation is involved.

In practice, graphics programmers use a relatively small set of simple, built-in transformations and build up the model transformations through a sequence of these simple transformations. Because each transformation works on the geometry it sees, we see the effect of the associative law for functions; in a piece of code represented by metacode such as

```
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

we see that `transformThree` is applied to the original geometry, `transformTwo` to the results of that transformation, and `transformOne` to the results of the second transformation. Letting `t1, t2,` and `t3` be the three transformations, respectively, we see by the application of the associative law for function composition that

```
t1(t2(t3(geometry))) = (t1*t2*t3)(geometry)
```

This shows us that in a product of transformations, applied by multiplying on the left, the transformation nearest the geometry is applied first, and that this principle extends across multiple transformations. This will be very important in the overall understanding of the overall order in which we operate on scenes, as we describe at the end of this section.

The modeling transformation for an object in a scene can change over time to create motion in a scene. For example, in a rigid-body animation, an object can be moved through the scene just by changing its model transformation between frames. This change can be made through standard built-in facilities in most graphics APIs, including OpenGL; we will discuss how this is done later.

3D world coordinate system

The 3D coordinate system shared by all the objects in the scene is called the *world coordinate system*. By considering every component of the scene as sharing this single world, we can treat the scene uniformly as we develop the presentation of the scene through the graphics display device to the user. The scene is a master design element that contains both the geometry of the objects placed in it and the geometry of lights that illuminate it. Note that the world coordinate system often is considered to have actual dimensions as it may well model some real-world environment. This coordinate system exists without any reference to a viewer, however; the viewer is added at the next stage.

3D eye coordinate system

Once the 3D world has been created, an application programmer would like the freedom to allow an audience to view it from any location. But graphics viewing models typically require a specific orientation and/or position for the eye at this stage. For example, the system might require that the eye position be at the origin, looking in $-Z$ (or sometimes $+Z$). So the next step in the pipeline is the *viewing transformation*, in which the coordinate system for the scene is changed to satisfy this

requirement. The result is the 3D eye coordinate system. We can think of this process as grabbing the arbitrary eye location and all the 3D world objects and sliding them around to realign the spaces so that the eye ends up at the proper place and looking in the proper direction. The relative positions between the eye and the other objects have not been changed; all the parts of the scene are simply anchored in a different spot in 3D space. Because standard viewing models may also specify a standard distance from the eyepoint to some fixed "look-at" point in the scene, there may also be some scaling involved in the viewing transformation. The viewing transformation is just a transformation in the same sense as modeling transformations, although it can be specified in a variety of ways depending on the graphics API. Because the viewing transformation changes the coordinates of the entire world space in order to move the eye to the standard position and orientation, we can consider the viewing transformation to be the inverse of whatever transformation placed the eye point in the position and orientation defined for the view. We will take advantage of this observation in the modeling chapter when we consider how to place the eye in the scene's geometry.

Clipping

At this point, we are ready to clip the object against the *3D viewing volume*. The viewing volume is the 3D volume that is determined by the projection to be used (see below) and that declares what portion of the 3D universe the viewer wants to be able to see. This happens by defining how much of the scene should be visible, and includes defining the left, right, bottom, top, near, and far boundaries of that space. Any portions of the scene that are outside the defined viewing volume are clipped and discarded. All portions that are inside are retained and passed along to the projection step. In Figure 0.2, it is clear that some of the world and some of the helicopter lie outside the viewable space, but note how the front of the image of the ground in the figure is clipped—is made invisible in the scene—because it is too close to the viewer's eye. This is a bit difficult to see, but look at the cliffs at the upper left of the scene to see a clipped edge.

Figure 0.2: Clipping on the Left, Bottom, and Right

Clipping is done as the scene is projected to the 2D eye coordinates in projections, as described next. Besides ensuring that the view includes only the things that should be visible, clipping also increases the efficiency of image creation because it eliminates some parts of the geometry from the rest of the display process.

Projections

The 3D eye coordinate system still must be converted into a 2D coordinate system before it can be mapped onto a graphics display device. The next stage of the pipeline performs this operation, called a *projection*. Before discussing the actual projection, we must think about what we will actually see in the graphic device. Imagine your eye placed somewhere in the scene, looking in a particular direction. You do not see the entire scene; you only see what lies in front of your eye and within your field of view. This space is called the *viewing volume* for your scene, and it includes a bit more than the eye point, direction, and field of view; it also includes a front plane, with the concept that you cannot see anything closer than this plane, and a back plane, with the concept that you cannot see anything farther than that plane. In Figure 0.3 we see two viewing volumes for the two kinds of projections that we will discuss in a moment.

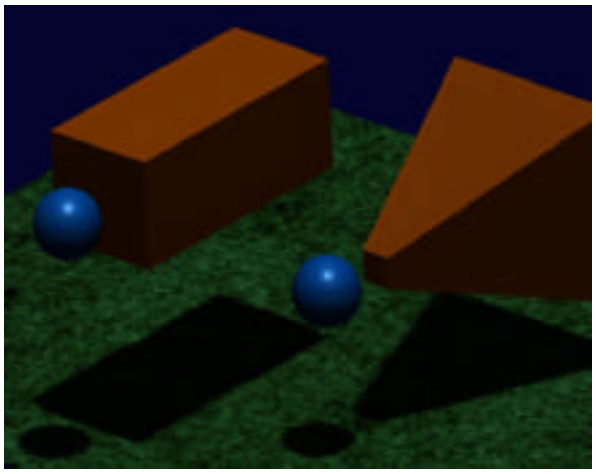


Figure 0.3: Parallel and Perspective Viewing Volumes, with Eyeballs

There are two kinds of projections commonly used in computer graphics. One maps all the points in the eye space to the viewing plane by simply ignoring the value of the z-coordinate, and as a result all points on a line parallel to the direction of the eye are mapped to the same point on the viewing plane. Such a projection is called a *parallel* projection, and it has the effect that the viewer can read accurate dimensions in the x- and y-coordinates. It is common for engineering drawings to present two parallel projections with the second including a 90° rotation of the world space so accurate z-coordinates can also be seen. The other projection acts as if the eye were a single point and each point in the scene is mapped along a line from the eye to that point, to a point on a plane in front of the eye, which is the classical technique of artists when drawing with perspective. Such a projection is called a *perspective* projection. And just as there are parallel and perspective projections, there are *parallel* (also called *orthographic*) and *perspective* viewing volumes. In a parallel projection, objects stay the same size as they get farther away. In a perspective projection, objects get smaller as they get farther away. Perspective projections tend to look more realistic, while parallel projections tend to make objects easier to line up. Each projection will display the geometry within the region of 3-space that is bounded by the right, left, top, bottom, back, and front planes described above. The region that is visible with each projection is often called its *view volume*. As we see in Figure 0.3, the viewing volume of a parallel projection is a rectangular region (here shown as a solid), while the viewing volume of a perspective projection has the shape of a pyramid that is truncated at the top (also shown as a solid). This kind of shape is a truncated pyramid and is sometimes called a *frustum*.

While the viewing volume describes the region in space that is included in the view, the actual view is what is displayed on the front clipping space of the viewing volume. This is a 2D space and is essentially the 2D eye space discussed below. Figure 0.4 presents a scene with both parallel and perspective projections; in this example, you will have to look carefully to see the differences!

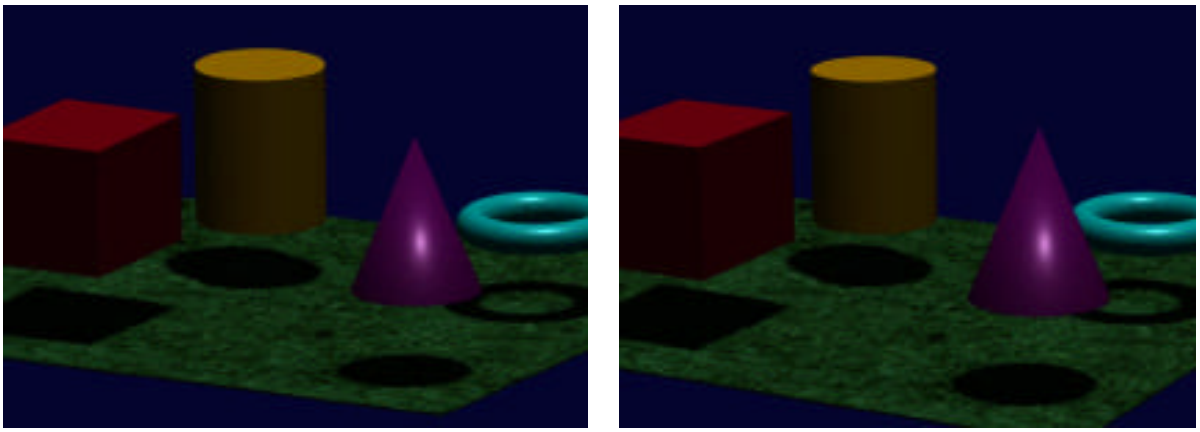


Figure 0.4: the same scene as presented by a parallel projection (left) and by a perspective projection (right)

2D eye coordinates

The space that projection maps to is a two-dimensional real-coordinate space that contains the geometry of the original scene after the projection is applied. Because a single point in 2D eye coordinates corresponds to an entire line segment in the 3D eye space, depth information is lost in the projection and it can be difficult to perceive depth, particularly if a parallel projection was used. Even in that case, however, if we display the scene with a hidden-surface technique, object occlusion will help us order the content in the scene. Hidden-surface techniques are discussed in a later chapter.

2D screen coordinates

The final step in the pipeline is to change units so that the object is in a coordinate system appropriate for the display device. Because the screen is a digital device, this requires that the real numbers in the 2D eye coordinate system be converted to integer numbers that represent screen coordinate. This is done with a proportional mapping followed by a truncation of the coordinate values. It is called the *window-to-viewport mapping*, and the new coordinate space is referred to as *screen coordinates*, or *display coordinates*. When this step is done, the entire scene is now represented by integer screen coordinates and can be drawn on the 2D display device.

Note that this entire pipeline process converts vertices, or *geometry*, from one form to another by means of several different transformations. These transformations ensure that the vertex geometry of the scene is consistent among the different representations as the scene is developed, but computer graphics also assumes that the *topology* of the scene stays the same. For instance, if two points are connected by a line in 3D model space, then those converted points are assumed to likewise be connected by a line in 2D screen space. Thus the geometric relationships (points, lines, polygons, ...) that were specified in the original model space are all maintained until we get to screen space, and are only actually implemented there.

Overall viewing process

Let's look at the overall operations on the geometry you define for a scene as the graphics system works on that scene and eventually displays it to your user. Referring again to Figure 0.1 and omitting the clipping and window-to-viewport process, we see that we start with geometry, apply the modeling transformation(s), apply the viewing transformation, and finally apply the projection to the screen. This can be expressed in terms of function composition as the sequence

```
projection(viewing(transformation(geometry)))
```

or, as we noted above with the associative law for functions and writing function composition as multiplication,

```
(projection * viewing * transformation) (geometry).
```

In the same way we saw that the operations nearest the geometry were performed before operations further from the geometry, then, we will want to define the projection first, the viewing next, and the transformations last before we define the geometry they are to operate on. We will see this sequence as a key factor in the way we structure a scene through the scene graph in the modeling chapter later in these notes.

Different implementation, same result

Warning! This discussion has shown the *concept* of how a vertex travels through the graphics pipeline. There are several ways of implementing this travel, any of which will produce a correct display. Do not be disturbed if you find out a graphics system does not manage the overall graphics pipeline process exactly as shown here. The basic principles and stages of the operation are still the same.

For example, OpenGL combines the modeling and viewing transformations into a single transformation known as the *modelview matrix*. This will force us to take a little different approach to the modeling and viewing process that integrates these two steps. Also, graphics hardware systems typically perform a window-to-normalized-coordinates operation prior to clipping so that hardware can be optimized around a particular coordinate system. In this case, everything else stays the same except that the final step would be normalized-coordinate-to-viewport mapping.

In many cases, we simply will not be concerned about the details of how the stages are carried out. Our goal will be to represent the geometry correctly at the modeling and world coordinate stages, to specify the eye position appropriately so the transformation to eye coordinates will be correct, and to define our window and projections correctly so the transformations down to 2D and to screen space will be correct. Other details will be left to a more advanced graphics course.

Summary of viewing advantages

One of the classic questions beginners have about viewing a computer graphics image is whether to use perspective or orthographic projections. Each of these has its strengths and its weaknesses. As a quick guide to start with, here are some thoughts on the two approaches:

Orthographic projections are at their best when:

- Items in the scene need to be checked to see if they line up or are the same size
- Lines need to be checked to see if they are parallel
- We do not care that distance is handled unrealistically
- We are not trying to move through the scene

Perspective projections are at their best when:

- Realism counts
- We want to move through the scene and have a view like a human viewer would have
- We do not care that it is difficult to measure or align things

In fact, when you have some experience with each, and when you know the expectations of the audience for which you're preparing your images, you will find that the choice is quite natural and will have no problem knowing which is better for a given image.

A basic OpenGL program

Our example programs that use OpenGL have some strong similarities. Each is based on the GLUT utility toolkit that usually accompanies OpenGL systems, so all the sample codes have this fundamental similarity. (If your version of OpenGL does not include GLUT, its source code is available online; check the page at

`http://www.reality.sgi.com/opengl/glut3/glut3.h`

and you can find out where to get it. You will need to download the code, compile it, and install it in your system.) Similarly, when we get to the section on event handling, we will use the MUI (micro user interface) toolkit, although this is not yet developed or included in this first draft release.

Like most worthwhile APIs, OpenGL is complex and offers you many different ways to express a solution to a graphical problem in code. Our examples use a rather limited approach that works well for interactive programs, because we believe strongly that graphics and interaction should be learned together. When you want to focus on making highly realistic graphics, of the sort that takes a long time to create a single image, then you can readily give up the notion of interactive work.

So what is the typical structure of a program that would use OpenGL to make interactive images? We will display this structure-only example in C, as we will with all our examples in these notes. OpenGL is not really compatible with the concept of object-oriented programming because it maintains an extensive set of state information that cannot be encapsulated in graphics classes, while object-oriented design usually calls for objects to maintain their own state. Indeed, as you will see when you look at the example programs, many functions such as event callbacks cannot even deal with parameters and must work with global variables, so the usual practice is to create a global application environment through global variables and use these variables instead of parameters to pass information in and out of functions. (Typically, OpenGL programs use side effects—passing information through external variables instead of through parameters—because graphics environments are complex and parameter lists can become unmanageable.)

In the code below, you will see that the main function involves mostly setting up the system. This is done in two ways: first, setting up GLUT to create and place the system window in which your work will be displayed, and second, setting up the event-handling system by defining the callbacks to be used when events occur. After this is done, main calls the main event loop that will drive all the program operations, as described in the chapter below on event handling.

The full code example that follows this outline also discusses many of the details of these functions and of the callbacks, so we will not go into much detail here. For now, the things to note are that the reshape callback sets up the window parameters for the system, including the size, shape, and location of the window, and defines the projection to be used in the view. This is called first when the main event loop is entered as well as when any window activity happens (such as resizing or dragging). The reshape callback requests a redisplay when it finishes, which calls the display callback function, whose task is to set up the view and define the geometry for the scene. When this is finished, OpenGL is finished and goes back to your computer system to see if there has been any other graphics-related event. If there has, your program should have a callback to manage it; if there has not, then the idle event is generated and the idle callback function is called; this may change some of the geometry parameters and then a redisplay is again called.

```
   #include <GL/glut.h>    // alternately "glut.h" for Macintosh
   // other includes as needed

// typedef and global data section
   // as needed

// function template section
   void doMyInit(void);
   void display(void);
   void reshape(int,int);
   void idle(void);
   // others as defined

// initialization function
   void doMyInit(void) {
      set up basic OpenGL parameters and environment
      set up projection transformation (ortho or perspective)
   }

// reshape function
   void reshape(int w, int h) {
      set up projection transformation with new window
         dimensions w and h
      post redisplay
   }

// display function
   void display(void){
      set up viewing transformation as in later chapters
      define the geometry, transformations, appearance you need
      post redisplay
   }

// idle function
   void idle(void) {
      update anything that changes between steps of the program
      post redisplay
   }

// other graphics and application functions
   // as needed

// main function -- set up the system, turn it over to events
   void main(int argc, char** argv) {
   // initialize system through GLUT and your own initialization
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
      glutInitWindowSize(windW,windH);
      glutInitWindowPosition(topLeftX,topLeftY);
      glutCreateWindow("A Sample Program");
      doMyInit();
   // define callbacks for events as needed; this is pretty minimal
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutIdleFunc(idle);
   // go into main event loop
      glutMainLoop();
   }
```

Now that we have seen a basic structure for an OpenGL program, we will present a complete, working program and will analyze the way it represents the graphics pipeline described earlier in this chapter, while describing the details of OpenGL that it uses. The program is the simple simulation of temperatures in a uniform metal bar that is described in the later chapter on graphical problem-solving in science, and we will only analyze the program structure, not its function. It creates the image shown in Figure 0.5.

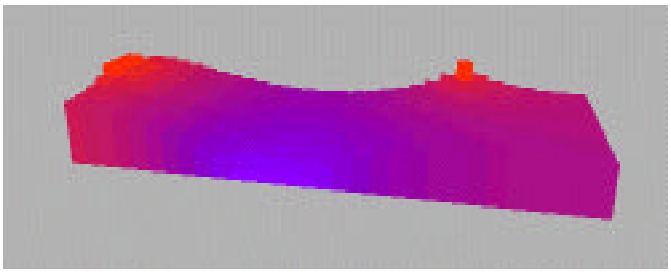


Figure 0.5: heat distribution in a bar

The code we will discuss is given below. We will segment it into components so you may see the different ways the individual pieces contribute to the overall graphics operations, and then we will discuss the pieces after the code listing.

```
// Example - heat flow in a thin rectangular body

// first section of code is declarations and initialization
// of the variables and of the system.
#include "glut.h" // <GL/glut.h> for windows
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define ROWS 10      //   body is ROWSxCOLS (unitless) squares
#define COLS 30
#define AMBIENT 25.0;//   ambient temperature, degrees Celsius
#define HOT  50.0    //   hot temperature of heat-source cell
#define COLD 0.0     //   cold temperature of cold-sink cell
#define NHOTS 4
#define NCOLDS 5

GLfloat angle = 0.0;
GLfloat temps[ROWS][COLS], back[ROWS+2][COLS+2];
GLfloat theta = 0.0, vp = 30.0;
// set locations of fixed hot and cold spots on the bar
int hotspots[NHOTS][2] =
     { {ROWS/2,0},{ROWS/2-1,0},{ROWS/2-2,0},{0,3*COLS/4} };
int coldspots[NCOLDS][2] =
     { {ROWS-1,COLS/3}, {ROWS-1,1+COLS/3}, {ROWS-1,2+COLS/3},
       {ROWS-1,3+COLS/3}, {ROWS-1,4+COLS/3} };
int myWin;

void myinit(void);
void cube(void);
void display(void);
void reshape(int, int);
void animate(void);
```

```
void myinit(void)
{
   int i,j;

   glEnable (GL_DEPTH_TEST);
   glClearColor(0.6, 0.6, 0.6, 1.0);

// set up initial temperatures in cells
   for (i=0; i<ROWS; i++) {
      for (j=0; j < COLS; j++) {
         temps[i][j] = AMBIENT;
      }
   }
   for (i=0; i<NHOTS; i++) {
      temps[hotspots[i][0]][hotspots[i][1]]=HOT; }
   for (i=0; i<NCOLDS; i++) {
      temps[coldspots[i][0]][coldspots[i][1]]=COLD; }
}
```

```
// This section creates a cube in model coordinates
// Unit cube in first octant
void cube (void)
{
   typedef GLfloat point [3];

   point v[8] = {
      {0.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0}, {0.0, 1.0, 1.0},
      {1.0, 0.0, 0.0}, {1.0, 0.0, 1.0},
      {1.0, 1.0, 0.0}, {1.0, 1.0, 1.0} };

   glBegin (GL_QUAD_STRIP);
      glVertex3fv(v[4]);
      glVertex3fv(v[5]);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
      glVertex3fv(v[6]);
      glVertex3fv(v[7]);
   glEnd();

   glBegin (GL_QUAD_STRIP);
      glVertex3fv(v[1]);
      glVertex3fv(v[3]);
      glVertex3fv(v[5]);
      glVertex3fv(v[7]);
      glVertex3fv(v[4]);
      glVertex3fv(v[6]);
      glVertex3fv(v[0]);
      glVertex3fv(v[2]);
   glEnd();
}
```

```
void display( void )
{
   #define SCALE 10.0
```

```
   int i,j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// This short section defines the viewing transformation
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   //           eye point      center of view       up
   gluLookAt(vp, vp/2., vp/4., 0.0, 0.0, 0.0,  0.0, 0.0, 1.0);

   // Draw the bars
   for (i = 0; i < ROWS; i++) {
      for (j = 0; j < COLS; j++) {
         glColor3f( temps[i][j]/HOT, 0.0, 1.0-temps[i][j]/HOT );
   // hotter redder; colder bluer
// Here is the modeling transformation for each item in the display
         glPushMatrix();
         glTranslatef((float)i-(float)ROWS/2.0,
                      (float)j-(float)COLS/2.0,0.0);
         // 0.1 cold, 4.0 hot
         glScalef(1.0, 1.0, 0.1+3.9*temps[i][j]/HOT);
         cube();
         glPopMatrix();
         }
      }
   glutSwapBuffers();
}

void reshape(int w,int h)
{
// This defines the projection transformation
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (float)w/(float)h, 1.0, 300.0);
}

void animate(void)
{
// This function is called whenever the system is idle; it makes
// changes in the data so that the next image is changed
   int i, j, m, n;
   float filter[3][3]={{ 0.    , 0.125, 0.    },
                       { 0.125 , 0.5,   0.125 },
                       { 0.    , 0.125, 0.    } };

   // increment temperatures throughout the material
   for (i=0; i<ROWS; i++) // backup temps up to recreate it
      for (j=0; j<COLS; j++)
         back[i+1][j+1] = temps[i][j]; // leave boundaries on back
   // fill boundaries with adjacent values from original temps[][]
   for (i=1; i<ROWS+2; i++) {
      back[i][0]=back[i][1];
      back[i][COLS+1]=back[i][COLS];
      }
   for (j=0; j<COLS+2; j++) {
```

```
      back[0][j] = back[1][j];
      back[ROWS+1][j]=back[ROWS][j];
      }
   for (i=0; i<ROWS; i++) // diffusion based on back values
      for (j=0; j<COLS; j++) {
         temps[i][j]=0.0;
         for (m=-1; m<=1; m++)
            for (n=-1; n<=1; n++)
               temps[i][j]+=back[i+1+m][j+1+n]*filter[m+1][n+1];
      }
   for (i=0; i<NHOTS; i++) {
      temps[hotspots[i][0]][hotspots[i][1]]=HOT; }
   for (i=0; i<NCOLDS; i++) {
      temps[coldspots[i][0]][coldspots[i][1]]=COLD; }

   // finished updating temps; now do the display
   glutPostRedisplay();
}

void main(int argc, char** argv)
{
// Initialize the GLUT system and define the window
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(50,50);
      myWin = glutCreateWindow("Temperature in bar");

      myinit();

// define the event callbacks and enter main event loop
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutIdleFunc(animate);
      glutMainLoop(); /* enter event loop */
}
```

The structure of the `main()` program using OpenGL

The `main()` program in an OpenGL-based application looks somewhat different from the programs we probably have seen before. This function has several key operations: it sets up the display mode, defines the window in which the display will be presented, and does whatever initialization is needed by the program. It then does something that may not be familiar to you: it defines a set of event callbacks, which are functions that are called by the system when an event occurs.

When you set up the display mode, you indicate to the system all the special features that your program will use at some point. In the example here,

```
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

you tell the system that you will be working in double-buffered mode, will use the RGB color model, and will be using depth testing. Some of these have to be enabled before they are actually used, as the depth testing is in the `myInit()` function with

```
      glEnable(GL_DEPTH_TEST).
```

Setting up the window (or windows—OpenGL will let you have multiple windows open) is handled by a set of GLUT calls that positions the window, defines the size of the window, and gives a title to the window. As the program runs, the window may be reshaped by the user, and this is handled by the `reshape()` function.

The way OpenGL handles event-driven programming is described in much more detail in a later chapter, but for now you need to realize that GLUT-based OpenGL (which is all we will describe in this book) operates entirely from events. For each event that the program is to handle, you need to define a callback function here in `main()`. When the main event loop is started, a reshape event is generated to create the window and a display event is created to draw an image in the window. If any other events have callbacks defined, then those callback functions are invoked when the events happen. The reshape callback allows you to move the window or change its size, and is called whenever you do any window manipulation. The idle callback allows the program to create a sequence of images by doing recomputations whenever the system is idle (is not creating an image or responding to another event), and then redisplaying the changed image.

Model space

The function `cube()` above defines a unit cube with sides parallel to the coordinate axes, one vertex at the origin, and one vertex at (1,1,1). This cube is created by defining an array of points that are the eight vertices of such a cube, and then using the `glBegin()...glEnd()` construction to draw the six squares that make up the cube through two quad strips. This is discussed in the chapter on modeling with OpenGL; for now, note that the cube uses its own set of coordinates that may or may not have anything to do with the space in which we will define our metallic strip to simulate heat transfer.

Modeling transformation

The modeling transformation is found in the `display()` function, and is quite simple: it defines the fundamental transformations that are to be applied to the basic geometry units as they are placed into the world. In our example, the basic geometry unit is a cube, and the cube is scaled in Z (but not in X or Y) and is then translated by X and Y (but not Z). The order of the transformations, the way each is defined, and the `pushMatrix()/popMatrix()` operations are described in the later chapter on modeling in OpenGL. For now it suffices to see that the transformations are defined in order to make a cube of the proper height to represent the temperature. If you were observant, you also noted that we also set the color for each cube based on the temperature.

3D world space

The 3D world space for this program is the space in which the graphical objects live after they have been placed by the modeling transformations. The translations give us one hint as to this space; we see that the x-coordinates of the translated cubes will lie between `-ROWS/2` and `ROWS/2`, while the y-coordinates of these cubes will lie between `-COLS/2` and `COLS/2`. Because `ROWS` and `COLS` are 30 and 10, respectively, the x-coordinates will lie between -15 and 15 and the y-coordinates will lie between -5 and 5. The low z-coordinate is 0 because that is never changed when the cubes are scaled, while the high z-coordinate is never larger then 4. Thus the entire bar lies in the region between -15 and 15 in x, -5 and 5 in y, and 0 and 4 in z. (Actually, this is not quite correct, but it is adequate for now; you are encouraged to find the small error.)

Viewing transformation

The viewing transformation is defined at the beginning of the `display()` function above. The code identifies that it is setting up the modelview matrix, sets that matrix to the identity (a

transformation that makes no changes to the world), and then specifies the view. A view is specified in OpenGL with the `gluLookAt()` call:

```
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
```

with parameters that include the coordinates of eye position (`ex, ey, ez`), the coordinates of the point at which the eye is looking (`lx, ly, lz`), and the coordinates of a vector that defines the "up" direction for the view (`ux, uy, uz`). This is discussed in more detail in the chapter below on viewing.

3D eye space

There is no specific representation of the 3D eye space in the program, because this is simply an intermediate stage in the production of the image. We can see, however, that we had set the center of view to the origin, which is the center of our image, and we had set our eye point to look at the origin from a point somewhat above and to the right of the center, so after the viewing transformation the object seems to be tilted up and to the side. This is the representation in the final 3D space that will be used to project the scene to the eye.

Projections

The projection operation is defined here in the `reshape()` function. It may be done in other places, but this is a good location and clearly separates the operation of projection from the operation of viewing.

Projections are specified fairly easily in the OpenGL system. An orthographic (or parallel) projection is defined with the function call:

```
glOrtho( left, right, bottom, top,  near, far );
```

where `left` and `right` are the x-coordinates of the left and right sides of the orthographic view volume, `bottom` and `top` are the y-coordinates of the bottom and top of the view volume, and `near` and `far` are the z-coordinates of the front and back of the view volume. A perspective projection can be defined with the function call:

```
gluPerspective( fovy, aspect, near, far );
```

In this function, the first parameter is the field of view in degrees, the second is the aspect ratio for the window, and the near and far parameters are as above. In this projection, it is assumed that your eye is at the origin so there is no need to specify the other four clipping planes; they are determined by the field of view and the aspect ratio.

When the window is reshaped, it is useful to take the width and height from the reshape event and define your projection to have the same aspect ratio (ratio of width to height) that the window has. That way there is no distortion introduced into the scene as it is seen through the newly-shaped window. If you use a fixed aspect ratio and change the window's shape, the original scene will be distorted to be seen through the new window, which can be confusing to the user.

2D eye space

This is the real 2D space to which the 3D world is projected, and it corresponds to the forward plane of the view volume. In order to provide uniform dimensions for mapping to the screen, the eye space is scaled so it has dimension -1 to 1 in each coordinate.

2D screen space

When the system was initialized, the window was defined to be 500x500 pixels with a top corner at (50,50). Thus the screen space is the set of pixels in that area of the screen. In fact, though, the window maintains its coordinate system independently of its location, so the point that had been

(0,0,0) in 3D eye space is now (249,249) in screen space. Note that screen space is discrete, not continuous, and its coordinates start at 0.

Another way to see the program

Another way to see how this program works is to consider it function-by-function instead of by the properties of the graphics pipeline. We will do this briefly here.

The task of the function `myinit()` is to set up the environment for the program so that the scene's fundamental environment is set up. This is a good place to compute values for arrays that define the geometry, to define specific named colors, and the like. At the end of this function you should set up the initial projection specifications.

The task of the function `display()` is to do everything needed to create the image. This can involve manipulating a significant amount of data, but the function does not allow any parameters. Here is the first place where the data for graphics problems must be managed through global variables. As we noted above, we treat the global data as a programmer-created environment, with some functions manipulating the data and the graphical functions using that data (the graphics environment) to define and present the display. In most cases, the global data is changed only through well-documented side effects, so this use of the data is reasonably clean. (Note that this argues strongly for a great deal of emphasis on documentation in your projects, which most people believe is not a bad thing.) Of course, some functions can create or receive control parameters, and it is up to you to decide whether these parameters should be managed globally or locally, but even in this case the declarations are likely to be global because of the wide number of functions that may use them. You will also find that your graphics API maintains its own environment, called its system state, and that some of your functions will also manipulate that environment, so it is important to consider the overall environment effect of your work.

The task of the function `reshape(int, int)` is to respond to user manipulation of the window in which the graphics are displayed. The two parameters are the width and height of the window in screen space (or in pixels) as it is resized by the user's manipulation, and should be used to reset the projection information for the scene. GLUT interacts with the window manager of the system and allows a window to be moved or resized very flexibly without the programmer having to manage any system-dependent operations directly. Surely this kind of system independence is one of the very good reasons to use the GLUT toolkit!

The task of the function `idle()` is to respond to the "idle" event — the event that nothing has happened. This function defines what the program is to do without any user activity, and is the way we can get animation in our programs. Without going into detail that should wait for our general discussion of events, the process is that the `idle()` function makes any desired changes in the global environment, and then requests that the program make a new display (with these changes) by invoking the function `glutPostRedisplay()` that simply requests the display function when the system can next do it by posting a "redisplay" event to the system.

The execution sequence of a simple program with no other events would then look something like is shown in Figure 0.6. Note that `main()` does not call the `display()` function directly; instead `main()` calls the event handling function `glutMainLoop()` which in turn makes the first call to `display()` and then waits for events to be posted to the system event queue. We will describe event handling in more detail in a later chapter.

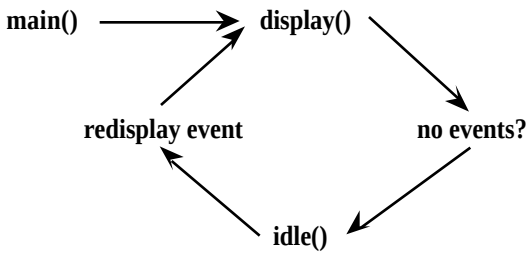


Figure 0.6: the event loop for the idle event

So we see that in the absence of any other event activity, the program will continues to apply the activity of the `idle()` function as time progresses, leading to an image that changes over time—that is, to an animated image.

Now that we have an idea of the graphics pipeline and know what a program can look like, we can move on to discuss how we specify the viewing and projection environment, how we define the fundamental geometry for our image, and how we create the image in the `display()` function with the environment that we define through the viewing and projection.

OpenGL extensions

In this chapter, and throughout these notes, we take a fairly limited view of the OpenGL graphics API. We do not work with most of the advanced features of the system and we only consider the more straightforward uses of the parts we cover. But OpenGL is capable of very sophisticated kinds of graphics, both in its original version and in versions that are available for specific kinds of graphics, and you should know of these because as you develop your graphics skills, you may find that the original "vanilla" OpenGL that we cover here will not do everything you want.

Advanced features of OpenGL include a number of special operation to store or manipulate information on a scene. These include modeling via polygon tessellation, NURBS surfaces, and defining and applying your own special-purpose transformations; the scissor test and the more general stencil buffer and stencil test; rendering in feedback mode to get details on what is being drawn; and facilities for client/server support. Remember that this is a general text, not a detailed presentation of OpenGL, and be ready to look further (see the references) for more information.

In addition to standard OpenGL, there are a number of extensions that support more specialized kinds of operations. These include the ARB imagine subset extension for image processing, the ARB multitexturing extension, vertex shader extensions, and many others. Some of these might have just the tools you need to do the very special things you want, so it would be useful for you to keep up to date on them. You can get information on extensions at the standard OpenGL Web site, `http://www.opengl.org`.

Chapter 1: Viewing and Projection

Prerequisites

An understanding of 2D and 3D geometry and familiarity with simple linear mappings.

Introduction

We emphasize 3D computer graphics consistently in these notes, because we believe that computer graphics should be encountered through 3D processes and that 2D graphics can be considered effectively as a special case of 3D graphics. But almost all of the viewing technologies that are readily available to us are 2D—certainly monitors, printers, video, and film—and eventually even the active visual retina of our eyes presents a 2D environment. So in order to present the images of the scenes we define with our modeling, we must create a 2D representation of the 3D scenes. As we saw in the graphics pipeline in the previous chapter, you begin by developing a set of models that make up the elements of your scene and set up the way the models are placed in the scene, resulting in a set of objects in a common world space. You then define the way the scene will be viewed and the way that view is presented on the screen. In this early chapter, we are concerned with the way we move from the world space to a 2D image with the tools of viewing and projection.

We set the scene for this process in the last chapter, when we defined the graphics pipeline. If we begin at the point where we have the 3D world coordinates—that is, where we have a complete scene fully defined in a 3D world—then this chapter is about creating a view of that scene in our display space of a computer monitor, a piece of film or video, or a printed page, whatever we want. To remind ourselves of the steps in this process, the pipeline is again shown in Figure 1.1.

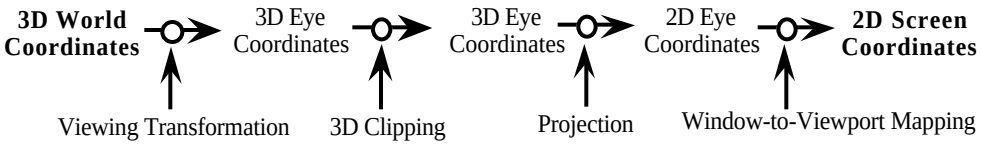


Figure 1.1: the graphics pipeline for creating an image of a scene

Let's consider an example of a world space and look at just what it means to have a view and a presentation of that space. One of the author's favorite places is Yosemite National Park, which is a wonderful example of a 3D world. Certainly there is a basic geometry in the park, made up of stone, wood, and water; this geometry can be seen from a number of points. In Figure 1.2 we see the classic piece of Yosemite geometry, the Half Dome monolith, from below in the valley and from above at Glacier Point. This gives us an excellent example of two views of the same geometry.

If you think about this area shown in these photographs, we can see the essential components of viewing. First, you notice that your view depends first on where you are standing. If you are standing on the valley floor, you see the face of the monolith in the classic view; if you are standing on the rim of Yosemite Valley at about the same height as Half Dome, you get a view that shows the profile of the rock. So your view depends on your position, which we call your *eye point.* Second, the view also depends on the point you are looking at, which we will call the *view reference point.* Both photos look generally towards the Half Dome monolith, or more specifically, towards a point in space behind the dome. This makes a difference not only in the view of the dome, but in the view of the region around the dome. In the classic Half Dome view from the valley, if you look off to the right you see the south wall of the valley; in the view from Glacier Point, if you look off to the right you see Vernal and Nevada falls on the Merced River

and, farther to the right, the high Sierra in the south of the park. The view also depends on the *breadth of field* of your view, whether you are looking at a wide part of the scene or a narrow part; again, the photograph at the left is a view of just Half Dome, while the one at the right is a panoramic view that includes the dome. While both photos are essentially square, you can visualize the left-hand photo as part of a photo that's vertical in layout while the right-hand photo looks more like it would come from a horizontal layout; this represents an *aspect ratio* for the image that can be part of its definition. Finally, although this may not be obvious at first because our minds process images in context, the view depends on your sense of the *up direction* in the scene: whether you are standing with your head upright or tilted (this might be easier to grasp if you think of the view as being defined by a camera instead of by your vision; it's clear that if you tilt a camera at a 45° angle you get a very different photo than one that's taken by a horizontal or vertical camera.) The world is the same in any case, but the determining factors for the image are where your eye is, the point you are looking toward, the breadth of your view, the aspect ratio of your view, and the way your view is tilted. All these will be accounted for when you define an image in computer graphics.

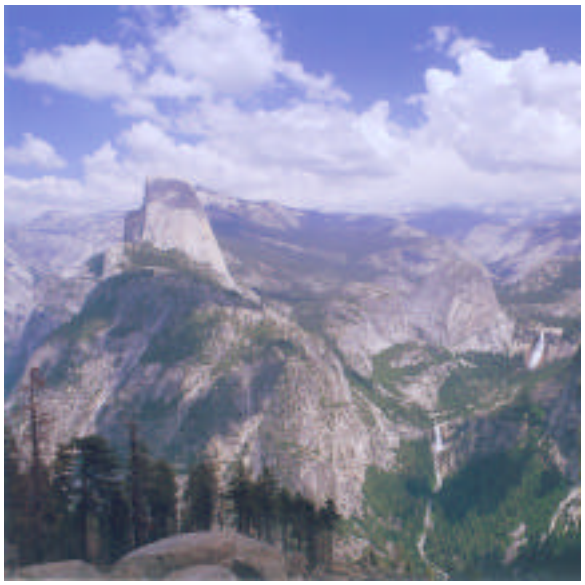



Figure 1.2: two photographs of Half Dome from different positions

Once you have determined your view, it must then be translated into an image that can be presented on your computer monitor. You may think of this in terms of recording an image on a digital camera, because the result is the same: each point of the view space (each pixel in the image) must be given a specific color. Doing that with the digital camera involves only capturing the light that comes through the lens to that point in the camera's sensing device, but doing it with computer graphics requires that we calculate exactly what will be seen at that particular point when the view is presented. We must define the way the scene is transformed into a two-dimensional space, which involves a number of steps: taking into account all the questions of what parts are in front of what other parts, what parts are out of view from the camera's lens, and how the lens gathers light from the scene to bring it into the camera. The best way to think about the lens is to compare two very different kinds of lenses: one is a wide-angle lens that gathers light in a very wide cone, and the other is a high-altitude photography lens that gathers light only in a very tight cylinder and processes light rays that are essentially parallel as they are transferred to the sensor. Finally, once the light from the continuous world comes into the camera, it is recorded on a digital sensor that only captures a discrete set of points.

This model of viewing is paralleled quite closely by a computer graphics system, and it follows the graphics pipeline that we discussed in the last chapter. You begin your work by modeling your scene in an overall world space (you may actually start in several modeling spaces, because you may model the geometry of each part of your scene in its own modeling space where it can be defined easily, then place each part within a single consistent world space to define the scene). This is very different from the viewing we discuss here but is covered in detail in the next chapter. The fundamental operation of viewing is to define an eye within your world space that represents the view you want to take of your modeling space. Defining the eye implies that you are defining a coordinate system relative to that eye position, and you must then transform your modeling space into a standard form relative to this coordinate system by defining, and applying, a viewing transformation. The fundamental operation of projection, in turn, is to define a plane within 3-dimensional space, define a mapping that projects the model into that plane, and displays that plane in a given space on the viewing surface (we will usually think of a screen, but it could be a page, a video frame, or a number of other spaces).

We will think of the 3D space we work in as the traditional X-Y-Z Cartesian coordinate space, usually with the X- and Y-axes in their familiar positions and with the Z-axis coming toward the viewer from the X-Y plane. This is a right-handed coordinate system, so-called because if you orient your right hand with your fingers pointing from the X-axis towards the Y-axis, your thumb will point towards the Z-axis. This orientation is commonly used for modeling in computer graphics because most graphics APIs define the plane onto which the image is projected for viewing as the X-Y plane, and project the model onto this plane in some fashion along the Z-axis. The mechanics of the modeling transformations, viewing transformation, and projection are managed by the graphics API, and the task of the graphics programmer is to provide the API with the correct information and call the API functionality in the correct order to make these operations work. We will describe the general concepts of viewing and projection below and will then tell you how to specify the various parts of this process to OpenGL.

Finally, it is sometimes useful to "cut away" part of an image so you can see things that would otherwise be hidden behind some objects in a scene. We include a brief discussion of clipping planes, a technique for accomplishing this action, because the system must clip away parts of the scene that are not visible in the final image.

Fundamental model of viewing

As a physical model, we can think of the viewing process in terms of looking through a rectangular frame that is held in front of your eye. You can move yourself around in the world, setting your eye into whatever position and orientation from you wish to see the scene. This defines your viewpoint and view reference point. The shape of the frame and the orientation you give it determine the aspect ratio and the up direction for the image. Once you have set your position in the world, you can hold up the frame to your eye and this will set your projection; by changing the distance of the frame from the eye you change the breadth of field for the projection. Between these two operations you define how you see the world in perspective through the hole. And finally, if you put a piece of transparent material that is ruled in very small squares behind the cardboard (instead of your eye) and you fill in each square to match the brightness you see in the square, you will create a copy of the image that you can take away from the original location. Of course, you only have a perspective projection instead of an orthogonal projection, but this model of viewing is a good place to start in understanding how viewing and projection work.

As we noted above, the goal of the viewing process is to rearrange the world so it looks as it would if the viewer's eye were in a standard position, depending on the API's basic model. When we define the eye location, we give the API the information it needs to do this rearrangement. In the next chapter on modeling, we will introduce the important concept of the *scene graph*, which

will integrate viewing and modeling. Here we give an overview of the viewing part of the scene graph.

The key point is that your view is defined by the location, direction, orientation, and field of view of the eye as we noted above. To understand this a little more fully, consider the situation shown in Figure 1.3. Here we have a world coordinate system that is oriented in the usual way, and within this world we have both a (simple) model and an eyepoint. At the eyepoint we have the coordinate system that is defined by the eyepoint-view reference point-up information that is specified for the view, so you may see the eyepoint coordinates in context. From this, you should try to visualize how the model will look once it is displayed with the view.

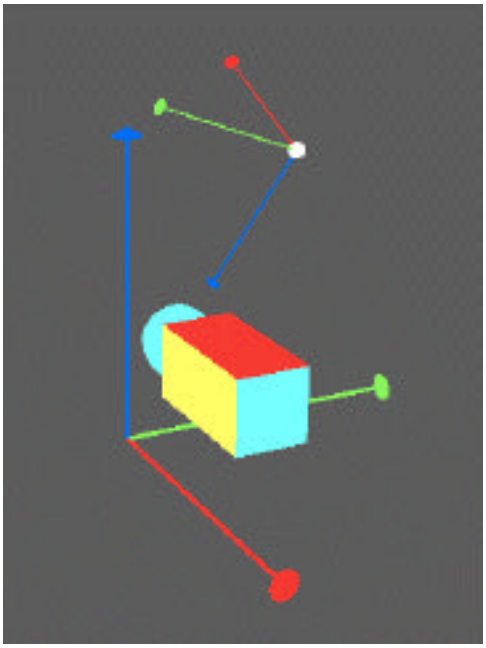


Figure 1.3: the eye coordinate system within the world coordinate system

In effect, you have defined a coordinate system within the world space relative to the eye. There are many ways to create this definition, but basically they all involve specifying three pieces of data in 3D space. Once this eye coordinate system is defined, we can apply an operation that changes the coordinates of everything in the world into equivalent representations in the eye coordinate system. This change of coordinates is a straightforward mathematical operation, performed by creating a change-of-basis matrix for the new system and then applying it to the world-space geometry. The transformation places the eye at the origin, looking along the *Z*-axis, and with the *Y*-axis pointed upwards; this view is similar to that shown in Figure 1.4. The specifications allow us to define the *viewing transformation* needed to move from the world coordinate system to the eye coordinate system. Once the eye is in standard position, and all your geometry is adjusted in the same way, the system can easily move on to project the geometry onto the viewing plane so the view can be presented to the user.

In the next chapter we will discuss modeling, and part of that process is using transformations to place objects that are defined in one position into a different position and orientations in world space. This can be applied to defining the eye point, and we can think of starting with the eye in standard position and applying transformations to place the eye where you want it. If we do that, then the viewing transformation is defined by computing the inverse of the transformation that placed the eye into the world. (If the concept of computing the inverse seems difficult, simply

think of undoing each of the pieces of the transformation; we will discuss this more in the chapter on modeling).

Once you have organized the viewing information as we have described, you must organize the information you send to the graphics system to define the way your scene is projected to the screen. The graphics system provides ways to define the projection and, once the projection is defined, the system will carry out the manipulations needed to map the scene to the display space. These operations will be discussed later in this chapter.

Definitions

There are a small number of things that you must consider when thinking of how you will view your scene. These are independent of the particular API or other graphics tools you are using, but later in the chapter we will couple our discussion of these points with a discussion of how they are handled in OpenGL. The things are:

- Your world must be seen, so you need to say how the view is defined in your model including the eye position, view direction, field of view, and orientation. This defines the viewing transformation that will be used to move from 3D world space to 3D eye space.
- In general, your world must be seen on a 2D surface such as a screen or a sheet of paper, so you must define how the 3D world is projected into a 2D space. This defines the 3D clipping and projection that will take the view from 3D eye space to 2D eye space.
- The region of the viewing device where the image is to be visible must be defined. This is the window, which should not be confused with the concept of window on your screen, though they often will both refer to the same space.
- When your world is seen in the window on the 2D surface, it must be seen at a particular place, so you must define the location where it will be seen. This defines the location of the viewport within the overall 2D viewing space and the window-to-viewport mapping that takes the 2D eye space to screen space.

We will call these three things setting up your viewing environment, defining your projection, and defining your window and viewport, respectively, and they are discussed in that order in the sections below.

Setting up the viewing environment

When you define a scene, you will want to do your work in the most natural world that would contain the scene, which we called the model space in the graphics pipeline discussion of the previous chapter. Objects defined in their individual model spaces are then placed in the world space with modeling transformations, as described in the next chapter on modeling. This world space is then transformed by the viewing transformation into a 3D space with the eye in standard position. To define the viewing transformation, you must set up a view by putting your eyepoint in the world space. This world is defined by the coordinate space you assumed when you modeled your scene as discussed earlier. Within that world, you define four critical components for your eye setup: where your eye is located, what point your eye is looking towards, how wide your field of view is, and what direction is vertical with respect to your eye. When these are defined to your graphics API, the geometry in your modeling is transformed with the viewing transformation to create the view as it would be seen with the environment that you defined.

A graphics API defines the computations that transform your geometric model as if it were defined in a standard position so it could be projected in a standard way onto the viewing plane. Each graphics API defines this standard position and has tools to create the transformation of your geometry so it can be viewed correctly. For example, OpenGL defines its viewing to take place in a right-handed coordinate system and transforms all the geometry in your scene (and we do mean *all* the geometry, including lights and directions, as we will see in later chapters) to place your eye

point at the origin, looking in the negative direction along the *Z*-axis. The eye-space orientation is illustrated in Figure 1.4.

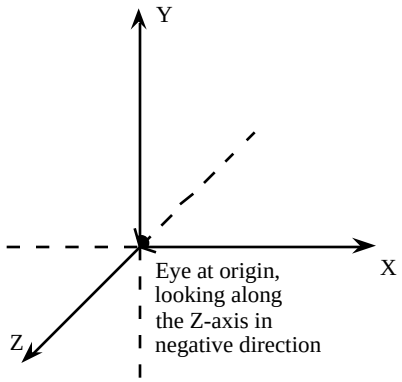


Figure 1.4: the standard OpenGL viewing model

Of course, no graphics API assumes that you can only look at your scenes with this standard view definition. Instead, you are given a way to specify your view very generally, and the API will convert the geometry of the scene so it is presented with your eyepoint in this standard position. This conversion is accomplished through the viewing transformation that is defined from your view definition as we discussed earlier.

The information needed to define your view includes your eye position (its (x,y,z) coordinates), the direction your eye is facing or the coordinates of a point toward which it is facing, and the direction your eye perceives as "up" in the world space. For example, the standard view that would be used unless you define another one has the position at the origin, or (0, 0, 0), the view direction or the "look-at" point coordinates as (0, 0, –1), and the up direction as (0, 1, 0). You will probably want to identify a different eye position for most of your viewing, because this is very restrictive and you probably will not want to define your whole viewable world as lying somewhere behind the *X*-*Y* plane. Your graphics API will give you a function that allows you to set your eye point as you desire.

The viewing transformation, then, is the transformation that takes the scene as you define it in world space and aligns the eye position with the standard model, giving you the eye space we discussed in the previous chapter. The key actions that the viewing transformation accomplishes are to rotate the world to align your personal up direction with the direction of the *Y*-axis, to rotate it again to put the look-at direction in the direction of the negative *Z*-axis (or to put the look-at point in space so it has the same *X*- and *Y*-coordinates as the eye point and a *Z*-coordinate less than the *Z*-coordinate of the eye point), to translate the world so that the eye point lies at the origin, and finally to scale the world so that the look-at point or look-at vector has the value (0, 0, –1). This is a very interesting transformation because what it *really* does is to invert the set of transformations that would move the eye point from its standard position to the position you define with your API function as above. This is very important in the modeling chapter below, and is discussed in some depth later in this chapter in terms of defining the view environment for the OpenGL API.

Defining the projection

The viewing transformation above defines the 3D eye space, but that cannot be viewed on our standard devices. In order to view the scene, it must be mapped to a 2D space that has some correspondence to your display device, such as a computer monitor, a video screen, or a sheet of paper. The technique for moving from the three-dimensional world to a two-dimensional world uses a projection operation that you define based on some straightforward fundamental principles.

When you (or a camera) view something in the real world, everything you see is the result of light that comes to the retina (or the film) through a lens that focuses the light rays onto that viewing surface. This process is a projection of the natural (3D) world onto a two-dimensional space. These projections in the natural world operate when light passes through the lens of the eye (or camera), essentially a single point, and have the property that parallel lines going off to infinity seem to converge at the horizon so things in the distance are seen as smaller than the same things when they are close to the viewer. This kind of projection, where everything is seen by being projected onto a viewing plane through or towards a single point, is called a *perspective projection*. Standard graphics references show diagrams that illustrate objects projected to the viewing plane through the center of view; the effect is that an object farther from the eye are seen as smaller in the projection than the same object closer to the eye.

On the other hand, there are sometimes situations where you want to have everything of the same size show up as the same size on the image. This is most common where you need to take careful measurements from the image, as in engineering drawings. An *orthographic projection* accomplishes this by projecting all the objects in the scene to the viewing plane by parallel lines. For orthographic projections, objects that are the same size are seen in the projection with the same size, no matter how far they are from the eye. Standard graphics texts contain diagrams showing how objects are projected by parallel lines to the viewing plane.

In Figure 1.5 we show two images of a wireframe house from the same viewpoint. The left-hand image of the figure is presented with a perspective projection, as shown by the difference in the apparent sizes of the front and back ends of the building, and by the way that the lines outlining the sides and roof of the building get closer as they recede from the viewer. The right-hand image of the figure is shown with an orthogonal projection, as shown by the equal sizes of the front and back ends of the building and the parallel lines outlining the sides and roof of the building. The differences between these two images is admittedly modest, but you should look for the differences noted above. It could be useful to use both projections on some of your scenes and compare the results to see how each of the projections works in different situations.

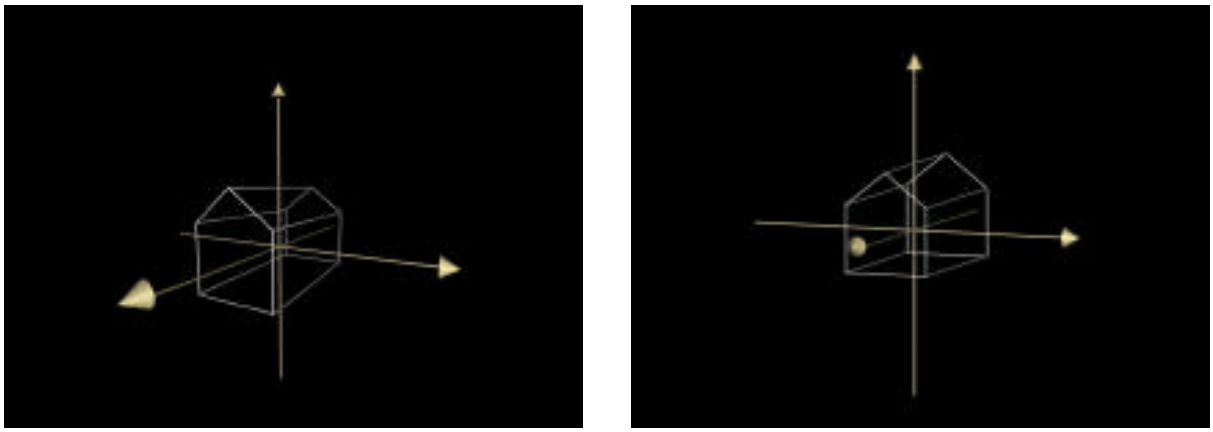


Figure 1.5: perspective image (left) and orthographic image (right) of a simple model

These two projections operate on points in 3D space in rather straightforward ways. For the orthographic projection, all points are projected onto the (X,Y)-plane in 3D eye space by simply omitting the Z-coordinate. Each point in 2D eye space is the image of a line parallel to the Z-axis, so the orthographic projection is sometimes called a *parallel projection*. For the perspective projection, any point is projected onto the plane $Z=1$ in 3D eye space at the point where the line from the point to the origin in 3D eye space meets that plane. Because of similar triangles, if the

point (x,y,z) is projected to the point (x',y'), we must have $x' = x/z$ and $(y' = y/z)$. Here each point in 2D eye space is the image of a line through that point and the origin in 3D eye space.

After a projection is applied, your scene is mapped to 2D eye space, as we discussed in the last chapter. However, the *z*-values in your scene are not lost. As each point is changed by the projection transformation, its *z*-value is retained for later computations such as depth tests or perspective-corrected textures. In some APIs such as OpenGL, the *z*-value is not merely retained but its sign is changed so that positive *z*-values will go away from the origin in a left-handed way. This convention allows the use of positive numbers in depth operations, which makes them more efficient.

View Volumes

A projection is often thought of in terms of its *view volume*, the region of space that is to be visible in the scene after the projection. With any projection, the fact that the projection creates an image on a rectangular viewing device implicitly defines a set of boundaries for the left, right, top, and bottom sides of the scene; these correspond to the left, right, top, and bottom of the viewing space. In addition, the conventions of creating images include not including objects that are too close to or too far from the eye point, and these give us the idea of front and back sides of the region of the scene that can be viewed. Overall, then, the projection defines a region in three-dimensional space that will contain all the parts of the scene that can be viewed. This region is called the *viewing volume* for the projection. The viewing volumes for the perspective and orthogonal projections are shown in Figure 1.6, with the eye point at the origin; this region is the space within the rectangular volume (left, for the orthogonal projection) or the pyramid frustum (right, for the perspective transformation). Note how these view volumes match the definitions of the regions of 3D eye space that map to points in 2D eye space, and note that each is presented in the left-handed viewing coordinate system described in Figure 1.4.

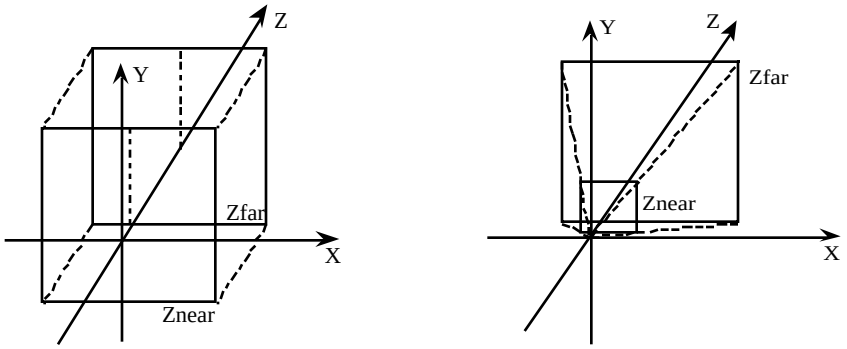


Figure 1.6: the viewing volumes for the orthogonal (left) and perspective (right) projections

While the orthographic view volume is defined only in a specified place in your model space, the orthogonal view volume may be defined wherever you need it because, being independent of the calculation that makes the world appear from a particular point of view, an orthogonal view can take in any part of space. This allows you to set up an orthogonal view of any part of your space, or to move your view volume around to view any part of your model. In fact, this freedom to place your viewing volume for the orthographic projection is not particularly important because you could always use simple translations to center the region you will see.

One of the reasons we pay attention to the view volume is that only objects that are inside the view volume for your projection will be displayed; anything else in the scene will be clipped, that is, be identified in the projection process as invisible, and thus will not be handled further by the graphics system. Any object that is partly within and partly outside the viewing volume will be clipped so

that precisely those parts inside the volume are seen, and we discuss the general concept and process of clipping later in this chapter. The sides of the viewing volume correspond to the projections of the sides of the rectangular space that is to be visible, but the front and back of the volume are less obvious—they correspond to the nearest and farthest space that is to be visible in the projection. These allow you to ensure that your image presents only the part of space that you want, and prevent things that might lie behind your eye from being projected into the visible space.

Calculating the perspective projection

The perspective projection is quite straightforward to compute, and although you do not need to carry this out yourself we will find it very useful later on to understand how it works. Given the general setup for the perspective viewing volume, let's look at a 2D version in Figure 1.7. Here we see that $Y/Y' = Z/1$, or simplifying, $Y' = Y/Z$. Thus with the conventions we have defined, the perspective projection defined on 3D eye space simply divides the original X and Y values by Z.

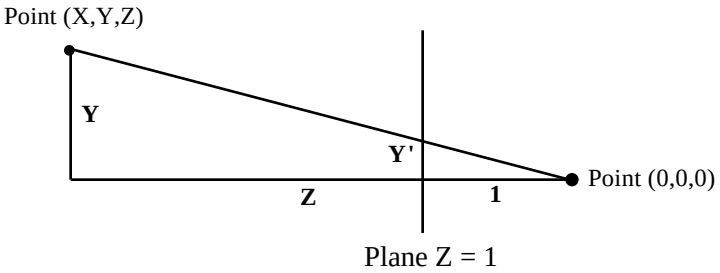


Figure 1.7: the perspective projection calculation

If we were to write this as a matrix, we would have the following:

$$\begin{matrix} 1/Z & 1 & 1 \\ 1 & 1/Z & 1 \\ 1 & 1 & 1 \end{matrix}$$

While this is a matrix, it is not a linear mapping, and that will have some significance later on when we realize that we must perform perspective corrections on some interpolations of object properties. Note here that we do not make any change in the value of Z, so that if we have the transformed values of X' and Y' and keep the original value of Z, we can reconstruct the original values as $X = X'*Z$ and $Y = Y'*Z$.

Defining the window and viewport

The scene as presented by the projection is still in 2D eye space, and the objects are all defined by real numbers. However, the display space is discrete, so the next step is a conversion of the geometry in 2D eye coordinates to discrete coordinates. This required identifying discrete screen points to replace the real-number eye geometry points, and introduces some sampling issues that must be handled carefully, but graphics APIs do this for you. The actual display space used depends on the window and the viewport you have defined for your image.

To a graphics system, a window is a rectangular region in your viewing space in which all of the drawing from your program will be done. It is usually defined in terms of the physical units of the drawing space. The window will be placed in your overall display device in terms of the device's coordinate system, which will vary between devices and systems. The window itself will have its own coordinate system, and the window space in which you define and manage your graphics

content will be called *screen space*, and is identified with integer coordinates. The smallest displayed unit in this space will be called a *pixel*, a shorthand for picture element. Note that the window for drawing is a distinct concept from the window in a desktop display window system, although the drawing window may in fact occupy a window on the desktop; we will be consistently careful to reserve the term window for the graphic display. While the window is placed in screen space, within the window itself—where we will do all our graphics work—we have a separate coordinate system that also has integer coordinates that represent pixel coordinates within the window itself. We will consistently think of the display space in terms of window points and window coordinates because they are all that matter to our image.

You will recall that we have a final transformation in the graphics pipeline from the 2D eye coordinate system to the 2D screen coordinate system. In order to understand that transformation, you need to understand the relation between points in two corresponding rectangular spaces. In this case, the rectangle that describes the scene to the eye is viewed as one space, and the rectangle on the screen where the scene is to be viewed is presented as another. The same processes apply to other situations that are particular cases of corresponding points in two rectangular spaces that we will see later, such as the relation between the position on the screen where the cursor is when a mouse button is pressed, and the point that corresponds to this in the viewing space, or points in the world space and points in a texture space.

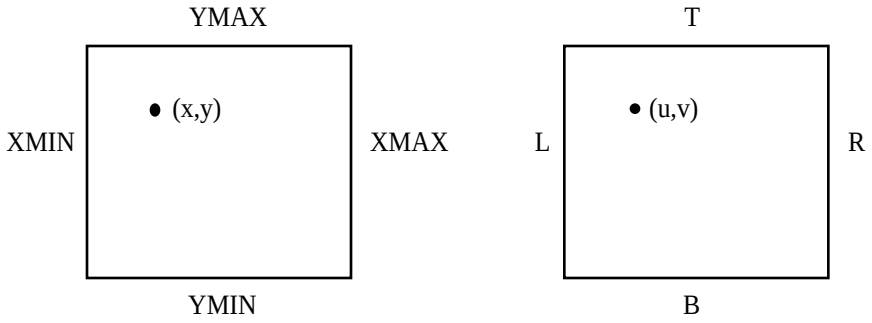


Figure 1.8: correspondences between points in two rectangles

In Figure 1.8, we show two rectangles with boundaries and points named as shown. In this example, we assume that the lower left corner of each rectangle has the smallest coordinate values in the rectangle. So the right-hand rectangle has a smallest X-value of L and a largest X-value of R, and a smallest Y-value of B and a largest Y-value of T, for example (think *left*, *right*, *top*, and *bottom* in this case).

With the names that are used in the figures, we have the proportions

$$X : XMIN :: XMAX : XMIN = u : L :: R : L$$
$$Y : YMIN :: YMAX : YMIN = v : B :: T : B$$

from which we can derive the equations:

$$(x - XMIN)/(XMAX - XMIN) = (u - L)/(R - L)$$
$$(y - YMIN)/(YMAX - YMIN) = (v - B)/(T - B)$$

and finally these two equations can be solved for the variables of either point in terms of the other, giving x and y in terms of u and v as:

$$x = XMIN + (u - L)(XMAX - XMIN)/(R - L)$$
$$y = YMIN + (v - B)(YMAX - YMIN)/(T - B)$$

or the dual equations that solve for (u,v) in terms of (x,y).

This discussion was framed in very general terms with the assumption that all our values are real numbers, because we were taking arbitrary ratios and treating them as exact values. This would

hold if we were talking about 2D eye space, but a moment's thought will show that these relations cannot hold in general for 2D screen space because integer ratios are only rarely exact. In the case of interest to us, one of these is 2D eye space and one is 2D screen space, so we must stop to ask how to modify our work for that case. For this case, to use the equations above for x and y we would regard the ratios for the right-hand rectangle in terms of real numbers and those for the left-hand rectangle as integers, we can get exact values for the ratios $(u-L)/(R-L)$ and $(v-B)/(T-B)$ and calculate real values for x and y, which we then truncate to get the desired integer values. This means that we take the view that an integer coordinate pair represents the unit square with that pair at the lower left of the square.

We noted that the window has a separate coordinate system, but we were not more specific about it. Your graphics API may use either of two conventions for window coordinates. The window may have its origin, or `(0,0)` value, at either the upper left or lower left corner. In the discussion above, we assumed that the origin was at the lower left because that is the standard mathematical convention, but graphics hardware often puts the origin at the top left because that corresponds to the lowest address of the graphics memory. If your API puts the origin at the upper left, you can make a simple change of variable as $Y' = YMAX - Y$ and using the Y' values instead of Y will put you back into the situation described in the figure.

When you create your image, you can choose to present it in a distinct sub-rectangle of the window instead of the entire window, and this part is called a *viewport*. A viewport is a rectangular region within that window to which you can restrict your image drawing. In any window or viewport, the ratio of its width to its height is called its *aspect ratio*. A window can have many viewports, even overlapping if needed to manage the effect you need, and each viewport can have its own image. Mapping an image to a viewport is done with exactly the same calculations we described above, except that the boundaries of the drawing area are the viewport's boundaries instead of the window's. The default behavior of most graphics systems is to use the entire window for the viewport. A viewport is usually defined in the same terms as the window it occupies, so if the window is specified in terms of physical units, the viewport probably will be also. However, a viewport can also be defined in terms of its size relative to the window, in which case its boundary points will be calculated from the window's.

If your graphics window is presented in a windowed desktop system, you may want to be able to manipulate your graphics window in the same way you would any other window on the desktop. You may want to move it, change its size, and click on it to bring it to the front if another window has been previously chosen as the top window. This kind of window management is provided by the graphics API in order to make the graphics window behavior on your system compatible with the behavior on all the other kinds of windows available. When you manipulate the desktop window containing the graphics window, the contents of the window need to be managed to maintain a consistent view. The graphics API tools will give you the ability to manage the aspect ratio of your viewports and to place your viewports appropriately within your window when that window is changed. If you allow the aspect ratio of a new viewport to be different than it was when defined, you will see that the image in the viewport seems distorted, because the program is trying to draw to the originally-defined viewport.

A single program can manage several different windows at once, drawing to each as needed for the task at hand. Individual windows will have different identifiers, probably returned when the window is defined, and these identifiers are used to specify which window will get the drawing commands as they are given. Window management can be a significant problem, but most graphics APIs have tools to manage this with little effort on the programmer's part, producing the kind of window you are accustomed to seeing in a current computing system—a rectangular space that carries a title bar and can be moved around on the screen and reshaped. This is the space in which all your graphical image will be seen. Of course, other graphical outputs such as video will

handle windows differently, usually treating the entire output frame as a single window without any title or border.

Some aspects of managing the view

Once you have defined the basic features for viewing your model, there are a number of other things you can consider that affect how the image is created and presented. We will talk about many of these over the next few chapters, but here we talk about hidden surfaces, clipping planes, and double buffering.

Hidden surfaces

Most of the things in our world are opaque, so we only see the things that are nearest to us as we look in any direction. This obvious observation can prove challenging for computer-generated images, however, because a graphics system simply draws what we tell it to draw in the order we tell it to draw them. In order to create images that have the simple "only show me what is nearest" property we must use appropriate tools in viewing our scene.

Most graphics systems have a technique that uses the geometry of the scene in order to decide what objects are in front of other objects, and can use this to draw only the part of the objects that are in front as the scene is developed. This technique is generally called Z-buffering because it uses information on the *z*-coordinates in the scene, as shown in Figure 1.4. In some systems it goes by other names; for example, in OpenGL this is called the *depth buffer*. This buffer holds the *z*-value of the nearest item in the scene for each pixel in the scene, where the *z*-values are computed from the eye point in eye coordinates. This *z*-value is the depth value after the viewing transformation has been applied to the original model geometry.

This depth value is not merely computed for each vertex defined in the geometry of a scene. When a polygon is processed by the graphics pipeline, an interpolation process is applied as described in the interpolation discussion in the chapter on the pipeline. This process will define a *z*-value, which is also the distance of that point from the eye in the *z*-direction, for each pixel in the polygon as it is processed. This allows a comparison of the *z*-value of the pixel to be plotted with the *z*-value that is currently held in the depth buffer. When a new point is to be plotted, the system first makes this comparison to check whether the new pixel is closer to the viewer than the current pixel in the image buffer and if it is, replaces the current point by the new point. This is a straightforward technique that can be managed in hardware by a graphics board or in software by simple data structures. There is a subtlety in this process that should be understood, however. Because it is more efficient to compare integers than floating-point numbers, the depth values in the buffer are kept as unsigned integers, scaled to fit the range between the near and far planes of the viewing volume with 0 as the front plane. If the near and far planes are far apart you may experience a phenomenon called "*Z*-fighting" in which roundoff errors when floating-point numbers are converted to integers causes the depth buffer shows inconsistent values for things that are supposed to be at equal distances from the eye. This problem is best controlled by trying to fit the near and far planes of the view as closely as possible to the actual items being displayed. This makes each integer value represent a smaller real number and so there is less likelihood of two real depths getting the same integer representation.

There are other techniques for ensuring that only the genuinely visible parts of a scene are presented to the viewer, however. If you can determine the depth (the distance from the eye) of each object in your model, then you may be able to sort a list of the objects so that you can draw them from back to front—that is, draw the farthest first and the nearest last. In doing this, you will replace anything that is hidden by other objects that are nearer, resulting in a scene that shows just the visible content. This is a classical technique called the *painter's algorithm* (because it mimics the way a painter could create an image using opaque paints) that was widely used in more limited

graphics systems, but it sometimes has real advantages over *Z*-buffering because it is faster (it doesn't require the pixel depth comparison for every pixel that is drawn) and because sometimes *Z*-buffering will give incorrect images, as we discuss when we discuss modeling transparency with blending in the color chapter. The painter's algorithm requires that you know the depth of each object in 3D eye space, however, and this can be difficult if your image includes moving parts or a moving eyepoint. Getting depths in eye space is discussed in the modeling chapter in the discussion of scene graphs.

Double buffering

A buffer is a set of memory that is used to store the result of computations, and most graphics APIs allow you to use two image buffers to store the results of your work. These are called the *color buffer* and the *back buffer*; the contents of the color buffer are what you see on your graphics screen. If you use only a single buffer, it is the color buffer, and as you generate your image, is written into the color buffer. Thus all the processes of clearing the buffer and writing new content to the buffer—new parts of your image—will all be visible to your audience.

Because it can take time to create an image, and it can be distracting for your audience to watch an image being built, it is unusual to use a single image buffer unless you are only creating one image. Most of the time you would use both buffers, and write your graphics to the back buffer instead of the color buffer. When your image is completed, you tell the system to switch the buffers so that the back buffer (with the new image) becomes the color buffer and the viewer sees the new image. When graphics is done this way, we say that we are using double buffering.

Because it can be disconcerting to actually watch the pixels changing as the image is created, particularly if you were creating an animated image by drawing one image after another, double buffering is essential to animated images. In fact, is used quite frequently for other graphics because it is more satisfactory to present a completed image instead of a developing image to a user. You must remember, however, that when an image is completed you must specify that the buffers are to be swapped, or the user will never see the new image!

Clipping planes

Clipping is the process of drawing with the portion of an image on one side of a plane drawn and the portion on the other side omitted. Recall from the discussion of geometric fundamentals that a plane is defined by a linear equation

$$Ax + By + Cz + D = 0$$

so it can be represented by the 4-tuple of real numbers (A,B,C,D). The plane divides the space into two parts: those points (x,y,z) for which $Ax + By + Cz + D$ is positive and those points for which it is negative. When you define the clipping plane for your graphics API with the functions it provides, you will probably specify it to the API by giving the four coefficients of the equation above. The operation of the clipping process is that any points for which this value is negative will not be displayed; any points for which it is positive or zero will be displayed.

Clipping defines parts of the scene that you do *not* want to display—parts that are to be left out for any reason. Any projection operation automatically includes clipping, because it must leave out objects in the space to the left, right, above, below, in front, and behind the viewing volume. In effect, each of the planes bounding the viewing volume for the projection is also a clipping plane for the image. You may also want to define other clipping planes for an image. One important reason to include clipping might be to see what is inside an object instead of just seeing the object's surface; you can define clipping planes that go through the object and display only the part of the object on one side or another of the plane. Your graphics API will probably allow you to define other clipping planes as well.

While the clipping process is handled for you by the graphics API, you should know something of the processes it uses. Because we generally think of graphics objects as built of polygons, the key point in clipping is to clip line segments (the boundaries of polygons) against the clipping plane. As we noted above, you can tell what side of a plane contains a point (x,y,z) by testing the algebraic sign of the expression $Ax + By + Cz + D$. If this expression is negative for both endpoints of a line segment, the entire line must lie on the "wrong" side of the clipping plane and so is simply not drawn at all. If the expression is positive for both endpoints, the entire line must lie on the "right" side and is drawn. If the expression is positive for one endpoint and negative for the other, then you must find the point for which the equation $Ax + By + Cz + D = 0$ is satisfied and then draw the line segment from that point to the point whose value in the expression is positive. If the line segment is defined by a linear parametric equation, the equation becomes a linear equation in one variable and so is easy to solve.

In actual practice, there are often techniques for handling clipping that are even simpler than that described above. For example, you might make only one set of comparisons to establish the relationship between a vertex of an object and a set of clipping planes such as the boundaries of a standard viewing volume. You would then be able to use these tests to drive a set of clipping operations on the line segment. We could then extend the work of clipping on line segments to clipping on the segments that are the boundaries of a polygon in order to clip parts of a polygon against one or more planes. We leave the details to the standard literature on graphics techniques.

Stereo viewing

Stereo viewing gives us an opportunity to see some of these viewing processes in action. Let us say quickly that this should not be your first goal in creating images; it requires a bit of experience with the basics of viewing before it makes sense. Here we describe binocular viewing—viewing that requires you to converge your eyes beyond the computer screen or printed image, but that gives you the full effect of 3D when the images are converged. Other techniques are described in later chapters.

Stereo viewing is a matter of developing two views of a model from two viewpoints that represent the positions of a person's eyes, and then presenting those views in a way that the eyes can see individually and resolve into a single image. This may be done in many ways, including creating two individual printed or photographed images that are assembled into a single image for a viewing system such as a stereopticon or a stereo slide viewer. (If you have a stereopticon, it can be very interesting to use modern technology to create the images for this antique viewing system!) Later in this chapter we describe how to present these as two viewports in a single window on the screen with OpenGL.

When you set up two viewpoints in this fashion, you need to identify two eye points that are offset by a suitable value in a plane perpendicular to the up direction of your view. It is probably simplest is you define your up direction to be one axis (perhaps the z-axis) and your overall view to be aligned with one of the axes perpendicular to that (perhaps the x-axis). You can then define an offset that is about the distance between the eyes of the observer (or perhaps a bit less, to help the viewer's eyes converge), and move each eyepoint from the overall viewpoint by half that offset. This makes it easier for each eye to focus on its individual image and let the brain's convergence create the merged stereo image. It is also quite important to keep the overall display small enough so that the distance between the centers of the images in the display is not larger than the distance between the viewer's eyes so that he or she can focus each eye on a separate image. The result can be quite startling if the eye offset is large so the pair exaggerates the front-to-back differences in the view, or it can be more subtle if you use modest offsets to represent realistic

views. Figure 1.9 shows the effect of such stereo viewing with a full-color shaded model. Later we will consider how to set the stereo eyepoints in a more systematic fashion.

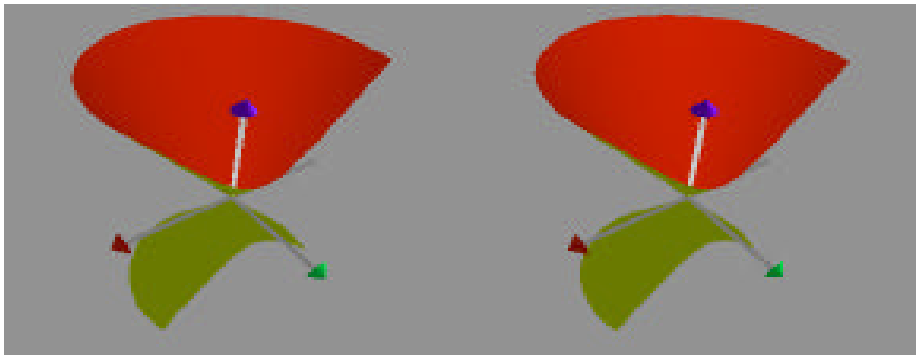


Figure 1.9: A stereo pair, including a clipping plane

Many people have physical limitations that do not allow their eyes to perform the kind of convergence that this kind of stereo viewing requires. Some people have general convergence problems which do not allow the eyes to focus together to create a merged image, and some simply cannot seem to see beyond the screen to the point where convergence would occur. In addition, if you do not get the spacing of the stereo pair right, or have the sides misaligned, or allow the two sides to refresh at different times, or ... well, it can be difficult to get this to work well for users. If some of your users can see the converged image and some cannot, that's probably as good as it's going to be.

There are other techniques for doing 3D viewing. When we discuss texture maps later, we will describe a technique that colors 3D images more red in the nearer parts and more blue in the more distant parts, as shown in Figure 1.10. This makes the images self-converge when you view them through a pair of ChromaDepth™ glasses, as we will describe there, so more people can see the spatial properties of the image, and it can be seen from anywhere in a room. There are also more specialized techniques such as creating alternating-eye views of the image on a screen with an overscreen that can be given alternating polarization and viewing them through polarized glasses that allow each eye to see only one screen at a time, or using dual-screen technologies such as head-mounted displays. The extension of the techniques above to these more specialized technologies is straightforward and is left to your instructor if such technologies are available.

Figure 1.10: a ChromaDepth™ display, courtesy of Michael J. Bailey, SDSC

Another technique involves preparing images in a way that will allow them to be picked up and displayed by special hardware such as the CrystalEyes® glasses with alternating eye vision from StereoGraphics Corp. There are a variety of ways that a left-eye and right-eye image can be combined so they can be picked up by special display hardware; these include side-by-side images, above-and-below images, or interlaced images. These combinations may require some distortion of the images that will have to be handled by the display hardware, as suggested by the distortions in the images in Figure 1.11 below, and the video stream may have to be manipulated in other ways to accommodate these approaches. In these cases, the display hardware manipulates the video output stream, separating the stream into two images that are displayed in synchronization with alternating polarized blanking of one eye, allowing the two eyes to see two distinct images and thus see the stereo pair naturally.

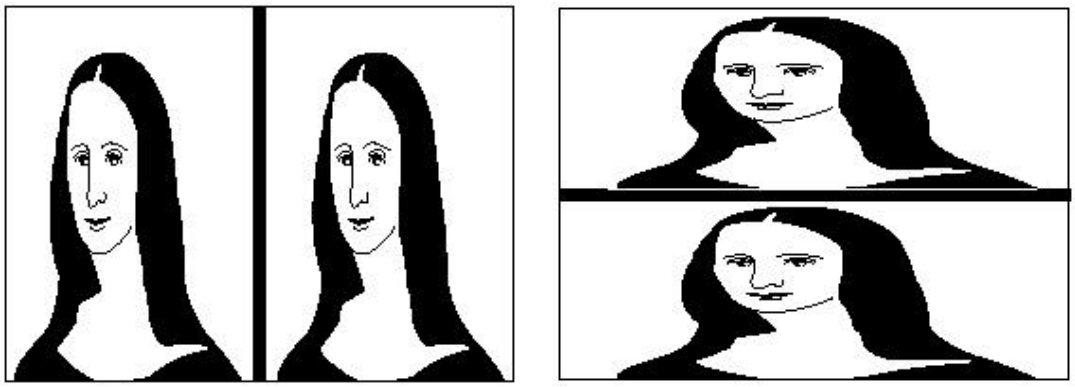


Figure 1.11: side-by-side images and above-below images, respectively.

Implementation of viewing and projection in OpenGL

The OpenGL code below captures much of the code needed in the discussion that follows in this section. It could be taken from a single function or could be assembled from several functions; in the sample structure of an OpenGL program in the previous chapter we suggested that the viewing and projection operations be separated, with the first part being at the top of the `display()` function and the latter part being at the end of the `init()` and `reshape()` functions.

```
// Define the projection for the scene
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);

// Define the viewing environment for the scene
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   //           eye point     center of view       up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

Defining a window and viewport: The window was defined in the previous chapter by a set of functions that initialize the window size and location and create the window. The details of window management are intentionally hidden from the programmer so that an API can work across many different platforms. In OpenGL, it is easiest to delegate the window setup to the GLUT toolkit where much of the system-dependent parts of OpenGL are defined; the functions to do this are:

```
glutInitWindowSize(width,height);
glutInitWindowPosition(topleftX,topleftY);
thisWindow = glutCreateWindow("Your window name here");
```

The integer value `thisWindow` that is returned by the glutCreateWindow can be used later to set the window you just created as the active window to which you will draw. This is done with the glutSetWindow function, as in

```
glutSetWindow(thisWindow);
```

which sets the window identified with `thisWindow` as the current window. If you are need to check which window is active, you can use the `glutGetWindow()` function that returns the window's value. In any case, no window is active until the main event loop is entered, as described in the previous chapter.

A viewport is defined by the `glViewport` function that specifies the lower left coordinates and the upper right coordinates for the portion of the window that will be used by the display. This function will normally be used in your initialization function for the program.

```
glViewport(VPLowerLeftX,VPLowerLeftY,VPUpperRightX,VPUpperRightY);
```

You can see the use of the viewport in the stereo viewing example below to create two separate images within one window.

Reshaping the window: The window is reshaped when it initially created or whenever is moved it to another place or made larger or smaller in any of its dimensions. These reshape operations are handled easily by OpenGL because the computer generates an event whenever any of these window reshapes happens, and there is an event callback for window reshaping. We will discuss events and event callbacks in more detail later, but the reshape callback is registered by the function `glutReshapeFunc(reshape)` which identifies a function `reshape(GLint w,GLint h)` that is to be executed whenever the window reshape event occurs and that is to do whatever is necessary to regenerate the image in the window.

The work that is done when a window is reshaped can involve defining the projection and the viewing environment and updating the definition of the viewport(s) in the window, or can delegate some of these to the display function. The reshape callback gets the dimensions of the window as it has been reshaped, and you can use these to control the way the image is presented in the reshaped window. For example, if you are using a perspective projection, the second parameter of the projection definition is the aspect ratio, and you can set this with the ratio of the width and height you get from the callback, as

```
gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);
```

This will let the projection compensate for the new window shape and retain the proportions of the original scene. On the other hand, if you really only want to present the scene in a given aspect ratio, then you can simply take the width and height and define a viewport in the window that has the aspect ratio you want. If you want a square presentation, for example, then simply take the smaller of the two values and define a square in the middle of the window as your viewport, and then do all your drawing to that viewport.

Any viewport you may have defined in your window probably needs either to be defined inside the reshape callback function so it can be redefined for resized windows or to be defined in the display function where the changed window dimensions can be taken into account when it is defined. The viewport probably should be designed directly in terms relative to the size or dimensions of the window, so the parameters of the reshape function should be used. For example, if the window is defined to have dimensions `(width, height)` as in the definition above, and if the viewport is to comprise the right-hand side of the window, then the viewport's coordinates are

```
(width/2, 0, width, height)
```

and the aspect ratio of the window is `width/(2*height)`. If the window is resized, you will probably want to make the width of the viewport no larger than the larger of half the new window width (to preserve the concept of occupying only half of the window) or the new window height

times the original aspect ratio. This kind of calculation will preserve the basic look of your images, even when the window is resized in ways that distort it far from its original shape.

Defining a viewing environment: To define what is usually called the viewing projection, you must first ensure that you are working with the `GL_MODELVIEW` matrix, then setting that matrix to be the identity, and finally define the viewing environment by specifying two points and one vector. The points are the eye point, the center of view (the point you are looking at), and the vector is the up vector—a vector that will be projected to define the vertical direction in your image. The only restrictions are that the eye point and center of view must be different, and the up vector must not be parallel to the vector from the eye point to the center of view. As we saw earlier, sample code to do this is:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
//            eye point     center of view       up
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

The `gluLookAt` function may be invoked from the `reshape` function, or it may be put inside the `display` function and variables may be used as needed to define the environment. In general, we will lean towards including the `gluLookAt` operation at the start of the `display` function, as we will discuss below. See the stereo view discussion below for an idea of what that can do.

The effect of the `gluLookAt(...)` function is to define a transformation that moves the eye point from its default position and orientation. That default position and orientation has the eye at the origin and looking in the negative z-direction, and oriented with the y-axis pointing upwards. This is the same as if we invoked the `gluLookAt` function with the parameters

```
gluLookAt(0., 0., 0., 0., 0., -1., 0., 1., 0.).
```

When we change from the default value to the general eye position and orientation, we define a set of transformations that give the eye point the position and orientation we define. The overall set of transformations supported by graphics APIs will be discussed in the modeling chapter, but those used for defining the eyepoint are:

1. a rotation about the *Z*-axis that aligns the *Y*-axis with the up vector,
2. a scaling to place the center of view at the correct distance along the negative *Z*-axis,
3. a translation that moves the center of view to the origin,
4. two rotations, about the *X*- and *Y*-axes, that position the eye point at the right point relative to the center of view, and
5. a translation that puts the center of view at the right position.

In order to get the effect you want on your overall scene, then, the viewing transformation must be the inverse of the transformation that placed the eye at the position you define, because it must act on all the geometry in your scene to return the eye to the default position and orientation. Because functions have the property that the inverse of a product is the product of the inverses in reverse order, as in

$$(f \quad g)^{-1} = g^{-1} \quad f^{-1}$$

for any *f* and *g*, the viewing transformation is built by inverting each of these five transformations in the reverse order. And because this must be done on all the geometry in the scene, it must be applied last, so it must be specified before any of the geometry is defined. Because of this we will usually see the `gluLookAt(...)` function as one of the first things to appear in the `display()` function, and its operation is the same as applying the transformations

1. translate the center of view to the origin,
2. rotate about the *X*- and *Y*-axes to put the eye point on the positive *Z*-axis,
3. translate to put the eye point at the origin,
4. scale to put the center of view at the point (0., 0., –1.), and
5. rotate around the *Z*-axis to restore the up vector to the *Y*-axis.

You may wonder why we are discussing at this point how the `gluLookAt(...)` function defines the viewing transformation that goes into the modelview matrix, but we will need to know about this later when we need to control the eye point as part of our modeling in more advanced kinds of scenes.

Defining perspective projection

A perspective projection is defined by first specifying that you want to work on the GL_PROJECTION matrix, and then setting that matrix to be the identity. You then specify the properties that will define the perspective transformation. In order, these are the field of view (an angle, in degrees, that defines the width of your viewing area), the aspect ratio (a ratio of width to height in the view; if the window is square this will probably be 1.0 but if it is not square, the aspect ratio will probably be the same as the ratio of the window width to height), the `zNear` value (the distance from the viewer to the plane that will contain the nearest points that can be displayed), and the `zFar` value (the distance from the viewer to the plane that will contain the farthest points that can be displayed). This sounds a little complicated, but once you've set it up a couple of times you'll find that it's very simple. It can be interesting to vary the field of view, though, to see the effect on the image.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
```

It is also possible to define your perspective projection by using the `glFrustum` function that defines the projection in terms of the viewing volume containing the visible items, as was shown in Figure 1.4 above. This call is

```
glFrustum( left, right,  bottom, top,  near, far );
```

Perhaps the gluPerspective function is more natural, so we will not discuss the glFrustum function further and leave it to the student who wants to explore it.

Defining an orthogonal projection: an orthogonal projection is defined much like a perspective projection except that the parameters of the projection itself are different. As you can see in the illustration of a parallel projection in Figure 1.3, the visible objects lie in a box whose sides are parallel to the *X*-, *Y*-, and *Z*-axes in the viewing space. Thus to define the viewing box for an orthogonal projection, we simply define the boundaries of the box as shown in Figure 1.3 and the OpenGL system does the rest.

```
glOrtho(xLow,xHigh,yLow,yHigh,zNear,zFar);
```

The viewing space is still the same left-handed space as noted earlier, so the `zNear` and `zFar` values are the distance from the *X*-*Y* plane in the negative direction, so that negative values of `zNear` and `zFar` refer to positions behind the eye (that is, in positive *Z*-space). There is no alternate to this function in the way that the `glFrustum(...)` is an alternative to the `gluLookAt(...)` function for parallel projections.

Managing hidden surface viewing

In the Getting Started chapter, we introduced the structure of a program that uses OpenGL and saw the `glutInitDisplayMode` function, called from `main`, is a way to define properties of the display. This function also allows the use of hidden surfaces if you specify `GLUT_DEPTH` as one of its parameters.

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

You must also enable the depth test. Enabling is a standard property of OpenGL; many capabilities of the system are only available after they are enabled through the `glEnable` function, as shown below.

```
glEnable(GL_DEPTH_TEST);
```

From that point the depth buffer is in use and you need not be concerned about hidden surfaces. OpenGL uses integer values for the depth test and so is vulnerable to *Z*-fighting. The default behavior of the depth test is that a point passes the depth test (and so is recorded in the scene) if its *z*-value is less than the *z*-value stored in the depth buffer, but this can be changed by using the `glDepthFunc(value)` function, where value is a symbolic constant. We will only use the depth test in its default form, but you can see OpenGL reference sources for more details.

If you want to turn off the depth test, there is a `glDisable` function as well as the `glEnable` function. Note the use of the enable and disable functions in enabling and disabling the clipping plane in the example code for stereo viewing.

Setting double buffering

Double buffering is a standard facility, and you will note that the function above that initializes the display mode includes a parameter `GLUT_DOUBLE` to set up double buffering. This indicates that you will use two buffers, called the *back buffer* and the *front buffer*, in your drawing. The content of the front buffer is displayed, and all drawing will take place to the back buffer. So in your `display()` function, you will need to call `glutSwapBuffers()` when you have finished creating the image; that will cause the back buffer to be exchanged with the front buffer and your new image will be displayed. An added advantage of double buffering is that there are a few techniques that use drawing to the back buffer and examination of that buffer's contents without swapping the buffers, so the work done in the back buffer will not be seen.

Defining clipping planes

In addition to the clipping OpenGL performs on the standard view volume in the projection operation, OpenGL allows you to define at least six clipping planes of your own, named `GL_CLIP_PLANE0` through `GL_CLIP_PLANE5`. The clipping planes are defined by the function `glClipPlane(plane, equation)` where `plane` is one of the pre-defined clipping planes above and `equation` is a vector of four `GLfloat` values. Once you have defined a clipping plane, it is enabled or disabled by a `glEnable(GL_CLIP_PLANEn)` function or equivalent `glDisable(...)` function. Clipping is performed when any modeling primitive is called when a clip plane is enabled; it is not performed when the clip plane is disabled. They are then enabled or disabled as needed to take effect in the scene. Specifically, some example code looks like

```
GLfloat myClipPlane[] = { 1.0, 1.0, 0.0, -1.0 };
glClipPlane(GL_CLIP_PLANE0, myClipPlane);
glEnable(GL_CLIP_PLANE0);
...
glDisable(GL_CLIP_PLANE0);
```

The stereo viewing example at the end of this chapter includes the definition and use of clipping planes.

Implementing a stereo view

In this section we describe the implementation of binocular viewing as described earlier in this chapter. The technique we will use is to generate two views of a single model as if they were seen from the viewer's separate eyes, and present these in two viewports in a single window on the screen. These two images are then manipulated together by manipulating the model as a whole, while viewer resolves these into a single image by focusing each eye on a separate image.

This latter process is fairly simple. First, create a window that is twice as wide as it is high, and whose overall width is twice the distance between your eyes. Then when you display your model,

do so twice, with two different viewports that occupy the left and right half of the window. Each display is identical except that the eye points in the left and right halves represent the position of the left and right eyes, respectively. This can be done by creating a window with space for both viewports with the window initialization function

```
#define W 600
#define H 300
width = W; height = H;
glutInitWindowSize(width,height);
```

Here the initial values set the width to twice the height, allowing each of the two viewports to be initially square. We set up the view with the overall view at a distance of `ep` from the origin in the *x*-direction and looking at the origin with the *z*-axis pointing up, and set the eyes to be at a given offset distance from the overall viewpoint in the *y*-direction. We then define the left- and right-hand viewports in the `display()` function as follows

```
//    left-hand viewport
      glViewport(0,0,width/2,height);
      ...
//                  eye point      center of view       up
      gluLookAt(ep, -offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
      ... code for the actual image goes here
      ...
//    right-hand viewport
      glViewport(width/2,0,width/2,height);
      ...
//                  eye point      center of view       up
      gluLookAt(ep,  offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
      ... the same code as above for the actual image goes here
      ...
```

This particular code example responds to a `reshape(width,height)` operation because it uses the window dimensions to set the viewport sizes, but it is susceptible to distortion problems if the user does not maintain the 2:1 aspect ratio as he or she reshapes the window. It is left to the student to work out how to create square viewports within the window if the window aspect ratio is changed.

Questions

This set of questions covers your recognition of issues in viewing and projections as you see them in your personal environment. They will help you see the effects of defining views and applying projections and the other topics in this chapter

1. Find a comfortable environment and examine the ways your view of that environment depend on your eyepoint and your viewing direction. Note how objects seem to move in front of and behind other objects as you move your eyepoing, and notice how objects move into the view from one side and out of the view on the other side as you rotate your viewing direction. (It may help if you make a paper or cardboard rectangle to look through as you do this.)

2. Because of the way our eyes work, we cannot see an orthogonal view of a scene. However, if we keep our eyes oriented in a fixed direction and move around in a scene, the view directly ahead of us will approximate a piece of an orthogonal view. For your familiar environment as above, try this and see if you can sketch what you see at each point and put them together into a single image.

3. Consider a painter's algorithm approach to viewing your environment; write down the objects you see in the order of farthest to nearest to your eye. Now move to another position in the environment and imagine drawing the things you see in the same order you wrote them down from the other viewpoint. What things are out of order and so would have the farther thing

drawn on top of the nearer thing? What conclusions can you draw about the calculations you would need to do for the painter's algorithm?

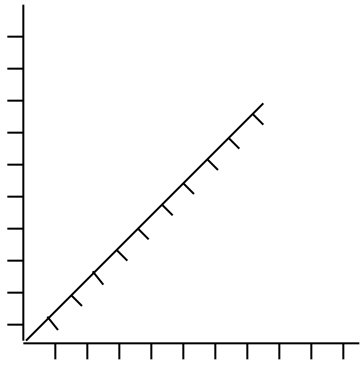
4. Imagine defining a plane through the middle of your environment so that everything on one side of the plane is not drawn. Make this plane go through some of the the objects you would see, so that one part of the object would be visible and another part invisible. What would the view of the environment look like? What would happen to the view if you switched the visibility of the two sides of the plane?

Exercises

These exercises ask you to carry out some calculations that are involved in creating a view of a scene and in doing the projection of the scene to the screen.

5. Take a standard perspective viewing definition with, say, a 45° field of view, an aspect ratio of 1.0, a distance to the front plane of the viewing frustum of 1.0, and a distance to the back plane of the viewing frustum of 20.0. For a point P=(x,y,1.) in the front plane, derive the parametric equation for the line segment within the frustum that all projects to P. Hint: the line segment goes through both the origin and P, and these two points can serve to define the segment's equation.

6. Create an X-Y-Z grid on a piece of paper using the convention that X is to the right, Y is up, and Z is into the page; an example is shown at the right.
   (a) In your familiar environment from question 1, place everything (or a number of things) from that environment into the grid with measured coordinates, to get familiar with coordinate systems to define positions. For this example, put everything into the space with non-negative coordinates (the first octant in 3D Cartesian coordinates) to make it easier to deal with the coordinates.
   (b) Define a position and direction for your eye in that same space and visualize what will be seen with that viewing definition, and then go to your space and see if your visualization was accurate. If not, then work out what was causing the inaccuracy.



7. In the numerically-modeled environment above, place your eyepoint in the (X,Z)-center of the space (the middle of the space left to right and front to back), and have your eye face the origin at floor height. Calculate the coordinates of each point in the space relative to the eye coordinate system, and try to identify a common process for each of these calculations.

Experiments

In these experiments, we will work with the very simple model of the house in Figure 1.5, though you are encouraged to replace that with a more interesting model of your own. The code for a function to create the house centered around the origin that you can call from your `display()` is given below to help you get started.

```
void drawHouse( void )
{
    point3 myHouse[10]={ { -1.0, -1.0,  2.0 }, { -1.0,  1.0,  2.0 },
                         {  0.0,  2.0,  2.0 }, {  1.0,  1.0,  2.0 },
                         {  1.0, -1.0,  2.0 }, { -1.0, -1.0, -2.0 },
                         { -1.0,  1.0, -2.0 }, {  0.0,  2.0, -2.0 },
                         {  1.0,  1.0, -2.0 }, {  1.0, -1.0, -2.0 }  };
    int i;

    glBegin(GL_LINE_STRIP);
      for ( i=0; i<5; i++)
        glVertex3fv(myHouse[i]);
      glVertex3fv(myHouse[0]);
    glEnd();
    glBegin(GL_LINE_STRIP);
      for ( i=0; i<5; i++)
        glVertex3fv(myHouse[i+5]);
      glVertex3fv(myHouse[5]);
    glEnd();
    for ( i=0; i<5; i++) {
      glBegin(GL_LINE_STRIP);
        glVertex3fv(myHouse[i]);
        glVertex3fv(myHouse[i+5]);
      glEnd();
    }
}
```

8. Create a program to draw the house with this function or to draw your own scene, and note what happens to the view as you move your eyepoint around the scene, always looking at the origin (0,0,0).

9. With the same program as above and a fixed eye point, experiment with the other parameters of the perspective view: the front and back view planes, the aspect ratio of the view, and the field of view of the projection. For each, note the effect so you can control these when you create more sophisticated images later.

Chapter 2: Principles of Modeling

Prerequisites

This chapter requires an understanding of simple 3-dimensional geometry, knowledge of how to represent points in 3-space, enough programming experience to be comfortable writing code that calls API functions to do required tasks, ability to design a program in terms of simple data structures such as stacks, and an ability to organize things in 3D space.

Introduction

Modeling is the process of defining the geometry that makes up a scene and implementing that definition with the tools of your graphics API. This chapter is critical in developing your ability to create graphical images and takes us from quite simple modeling to fairly complex modeling based on hierarchical structures, and discusses how to implement each of these different stages of modeling in OpenGL. It is fairly comprehensive for the kinds of modeling one would want to do with a basic graphics API, but there are other kinds of modeling used in advanced API work and some areas of computer graphics that involve more sophisticated kinds of constructions than we include here, so we cannot call this a genuinely comprehensive discussion. It is, however, a good enough introduction to give you the tools to start creating interesting images.

The chapter has three distinct parts because there are three distinct levels of modeling that you will use to create images. We begin with simple geometric modeling: modeling where you define the coordinates of each vertex of each component you will use at the point where that component will reside in the final scene. This is straightforward but can be very time-consuming to do for complex scenes, so we will also discuss importing models from various kinds of modeling tools that can allow you to create parts of a scene more easily.

The second section describes the next step in modeling: extending the simple objects defined in standard positions so they can have any size, any orientation, and any position. This extends the utility of your simple modeling by providing a set of primitive transformations you can apply to your simple geometry in order to create more general model components in your scene. This is a very important part of the modeling process because it allows you to use appropriate transformations that allow you to start with standard templates for a modest number of kinds of graphic objects and then generalize them and place them in your scene as needed. These transformations are also critical to the ability to define and implement motion in your scenes because it is typical to move objects, lights, and the eyepoint with transformations that are controlled by parameters that change with time. This can allow you to extend your modeling to define animations that can represent such concepts as changes over time.

In the third section of the chapter we give you an important tool to organize complex images by introducing the concept of the scene graph, a modeling tool that gives you a unified approach to defining all the objects and transformations that are to make up a scene and to specifying how they are related and presented. We then describe how you work from the scene graph to write the code that implements your model. This concept is new to the introductory graphics course but has been used in some more advanced graphics tools, and we believe you will find it to make the modeling process much more straightforward for anything beyond a very simple scene. In the second level of modeling discussed in this section, we introduce hierarchical modeling in which objects are designed by assembling other objects to make more complex structures. These structures can allow you to simulate actual physical assemblies and develop models of structures like physical machines. Here we develop the basic ideas of scene graphs introduced earlier to get a structure that allows individual components to move relative to each other in ways that would be difficult to define from first principles.

Simple Geometric Modeling

Introduction

Computer graphics deals with geometry and its representation in ways that allow it to be manipulated and displayed by a computer. Because these notes are intended for a first course in the subject, you will find that the geometry will be simple and will use familiar representations of 3-dimensional space. When you work with a graphics API, you will need to work with the kinds of object representations that API understands, so you must design your image or scene in ways that fit the API's tools. For most APIs, this means using only a few simple graphics primitives, such as points, line segments, and polygons.

The space we will use for our modeling is simple Euclidean 3-space with standard coordinates, which we will call the *X*-, *Y*-, and *Z*-coordinates. Figure 2.1 below illustrates a point, a line segment, a polygon, and a polyhedron—the basic elements of the computer graphics world that you will use for most of your graphics. In this space a *point* is simply a single location in 3-space, specified by its coordinates and often seen as a triple of real numbers such as (*px*, *py*, *pz*). A point is drawn on the screen by lighting a single pixel at the screen location that best represents the location of that point in space. To draw the point you will specify that you want to draw points and specify the point's coordinates, usually in 3-space, and the graphics API will calculate the coordinates of the point on the screen that best represents that point and will light that pixel. Note that a point is usually presented as a square, not a dot, as indicated in the figure. A *line segment* is determined by its two specified endpoints, so to draw the line you indicate that you want to draw lines and define the points that are the two endpoints. Again, these endpoints are specified in 3-space and the graphics API calculates their representations on the screen, and draws the line segment between them. A *polygon* is a region of space that lies in a plane and is bounded in the plane by a collection of line segments. It is determined by a sequence of points (called the *vertices* of the polygon) that specify a set of line segments that form its boundary, so to draw the polygon you indicate that you want to draw polygons and specify the sequence of vertex points. A *polyhedron* is a region of 3-space bounded by polygons, called the faces of the polyhedron. A polyhedron is defined by specifying a sequence of faces, each of which is a polygon. Because figures in 3-space determined by more than three vertices cannot be guaranteed to line in a plane, polyhedra are often defined to have triangular faces; a triangle always lies in a plane (because three points in 3-space determine a plane. As we will see when we discuss lighting and shading in subsequent chapters, the direction in which we go around the vertices of each face of a polygon is very important, and whenever you design a polyhedron, you should plan your polygons so that their vertices are ordered in a sequence that is counterclockwise as seen from outside the polyhedron (or, to put it another way, that the angle to each vertex as seen from a point inside the face is increasing rather than decreasing as you go around each face).

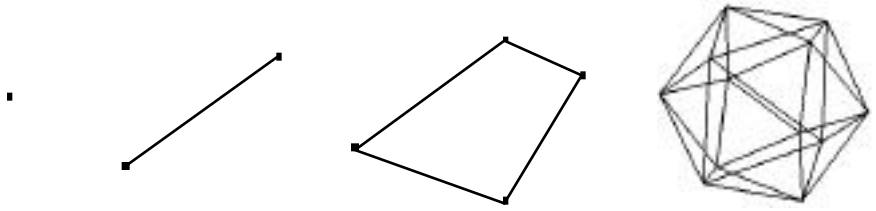


Figure 2.1: a point, a line segment, a polygon, and a polyhedron

Before you can create an image, you must define the objects that are to appear in that image through some kind of modeling process. Perhaps the most difficult—or at least the most time-consuming—part of beginning graphics programming is creating the models that are part of the

image you want to create. Part of the difficulty is in designing the objects themselves, which may require you to sketch parts of your image by hand so you can determine the correct values for the points used in defining it, for example, or it may be possible to determine the values for points from some other technique. Another part of the difficulty is actually entering the data for the points in an appropriate kind of data structure and writing the code that will interpret this data as points, line segments, and polygons for the model. But until you get the points and their relationships right, you will not be able to get the image right.

Definitions

We need to have some common terminology as we talk about modeling. We will think of modeling as the process of defining the objects that are part of the scene you want to view in an image. There are many ways to model a scene for an image; in fact, there are a number of commercial programs you can buy that let you model scenes with very high-level tools. However, for much graphics programming, and certainly as you are beginning to learn about this field, you will probably want to do your modeling by defining your geometry in terms of relatively simple primitive terms so you may be fully in control of the modeling process.

Besides defining a single point, line segment, or polygon, graphics APIs provide modeling support for defining larger objects that are made up of several simple objects. These can involve disconnected sets of objects such as points, line segments, quads, or triangles, or can involve connected sets of points, such as line segments, quad strips, triangle strips, or triangle fans. This allows you to assemble simpler components into more complex groupings and is often the only way you can define polyhedra for your scene. Some of these modeling techniques involve a concept called *geometry compression,* which allow you to define a geometric object using fewer vertices than would normally be needed. The OpenGL support for geometry compression will be discussed as part of the general discussion of OpenGL modeling processes. The discussions and examples below will show you how to build your repertoire of techniques you can use for your modeling.

Before going forward, however, we need to mention another way to specify points for your models. In some cases, it can be helpful to think of your 3-dimensional space as embedded as an affine subspace of 4-dimensional space. If we think of 4-dimensional space as having X, Y, Z, and W components, this embedding identifies the three-dimensional space with the subspace $W=1$ of the four-dimensional space, so the point (x, y, z) is identified with the four-dimensional point $(x, y, z, 1)$. Conversely, the four-dimensional point (x, y, z, w) is identified with the three-dimensional point $(x/w, y/w, z/w)$ whenever $w \neq 0$. The four-dimensional representation of points with a non-zero `w` component is called *homogeneous coordinates*, and calculating the three-dimensional equivalent for a homogeneous representation by dividing by `w` is called *homogenizing* the point. When we discuss transformations, we will sometimes think of them as 4x4 matrices because we will need them to operate on points in homogeneous coordinates.

Not all points in 4-dimensional space can be identified with points in 3-space, however. The point $(x, y, z, 0)$ is not identified with a point in 3-space because it cannot be homogenized, but it is identified with the direction defined by the vector $\langle x, y, z \rangle$. This can be thought of as a "point at infinity" in a certain direction. This has an application in the chapter below on lighting when we discuss directional instead of positional lights, but in general we will not encounter homogeneous coordinates often in these notes.

Some examples

In this section we will describe the kinds of simple objects that are directly supported by most graphics APIs. We begin with very simple objects and proceed to more complex ones, but you

will find that both simple and complex objects will be needed in your work. With each kind of primitive object, we will describe how that object is specified, and in later examples, we will create a set of points and will then show the function call that draws the object we have defined.

Point and points

To draw a single point, we will simply define the coordinates of the point and give them to the graphics API function that draws points. Such a function can typically handle one point or a number of points, so if we want to draw only one point, we provide only one vertex; if we want to draw more points, we provide more vertices. Points are extremely fast to draw, and it is not unreasonable to draw tens of thousands of points if a problem merits that kind of modeling. On a very modest-speed machine without any significant graphics acceleration, a 50,000 point model can be re-drawn in a small fraction of a second.

Line segments

To draw a single line segment, we must simply supply two vertices to the graphics API function that draws lines. Again, this function will probably allow you to specify a number of line segments and will draw them all; for each segment you simply need to provide the two endpoints of the segment. Thus you will need to specify twice as many vertices as the number of line segments you wish to produce.

The simple way that a graphics API handles lines hides an important concept, however. A line is a continuous object with real-valued coordinates, and it is displayed on a discrete object with integer screen coordinates. This is, of course, the difference between model space and eye space on one hand and screen space on the other. While we focus on geometric thinking in terms that overlook the details of conversions from eye space to screen space, you need to realize that algorithms for such conversions lie at the foundation of computer graphics and that your ability to think in higher-level terms is a tribute to the work that has built these foundations.

Connected lines

Connected lines—collections of line segments that are joined "head to tail" to form a longer connected group—are shown in Figure 2.2. These are often called line strips and line loops, and your graphics API will probably provide a function for drawing them. The vertex list you use will define the line segments by using the first two vertices for the first line segment, and then by using

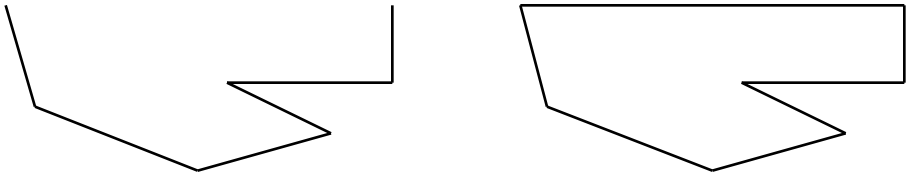


Figure 2.2: a line strip and a line loop

each new vertex and its predecessor to define each additional segment. The difference between a line strip and a line loop is that the former does not connect the last vertex defined to the first vertex, leaving the figure open; the latter includes this extra segment and creates a closed figure. Thus the number of line segments drawn by the a line strip will be one fewer than the number of vertices in the vertex list, while a line loop will draw the same number of segments as vertices. This is a geometry compression technique because to define a line strip with N segments you only specify $N+1$ vertices instead of $2N$ vertices; instead of needing to define two points per line segment, each segment after the first only needs one vertex to be defined.

Triangle

To draw one or more unconnected triangles, your graphics API will provide a simple triangle-drawing function. With this function, each set of three vertices will define an individual triangle so that the number of triangles defined by a vertex list is one third the number of vertices in the list. The humble triangle may seem to be the most simple of the polygons, but as we noted earlier, it is probably the most important because no matter how you use it, and no matter what points form its vertices, it always lies in a plane. Because of this, most polygon-based modeling really comes down to triangle-based modeling in the end, and almost every kind of graphics tool knows how to manage objects defined by triangles. So treat this humblest of polygons well and learn how to think about polygons and polyhedra in terms of the triangles that make them up.

Sequence of triangles

Triangles are the foundation of most truly useful polygon-based graphics, and they have some very useful capabilities. Graphics APIs often provide two different geometry-compression techniques to assemble sequences of triangles into your image: triangle strips and triangle fans. These techniques can be very helpful if you are defining a large graphic object in terms of the triangles that make up its boundaries, when you can often find ways to include large parts of the object in triangle strips and/or fans. The behavior of each is shown in Figure 2.3 below. Note that this figure and similar figures that show simple geometric primitives are presented as if they were drawn in 2D space. In fact they are not, but in order to make them look three-dimensional we would need to use some kind of shading, which is a separate concept discussed in a later chapter (and which is used to present the triangle fan of Figure 2.18). We thus ask you to think of these as three-dimensional, even though they look flat.

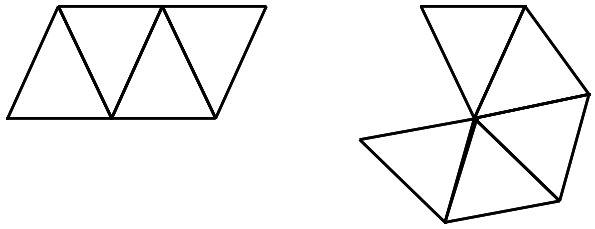


Figure 2.3: triangle strip and triangle fan

Most graphics APIs support both techniques by interpreting the vertex list in different ways. To create a triangle strip, the first three vertices in the vertex list create the first triangle, and each vertex after that creates a new triangle with the two vertices immediately before it. We will see in later chapters that the order of points around a polygon is important, and we must point out that these two techniques behave quite differently with respect to polygon order; for triangle fans, the orientation of all the triangles is the same (clockwise or counterclockwise), while for triangle strips, the orientation of alternate triangles is reversed. This may require some careful coding when lighting models are used. To create a triangle fan, the first three vertices create the first triangle and each vertex after that creates a new triangle with the point immediately before it and the first point in the list. In each case, the number of triangles defined by the vertex list is two less than the number of vertices in the list, so these are very efficient ways to specify triangles.

Quadrilateral

A convex quadrilateral, often called a “quad” to distinguish it from a general quadrilateral because the general quadrilateral need not be convex, is any convex 4-sided figure. The function in your graphics API that draws quads will probably allow you to draw a number of them. Each

quadrilateral requires four vertices in the vertex list, so the first four vertices define the first quadrilateral, the next four the second quadrilateral, and so on, so your vertex list will have four times as many points as there are quads. The sequence of vertices is that of the points as you go around the perimeter of the quadrilateral. In an example later in this chapter, we will use six quadrilaterals to define a cube that will be used in later examples.

Sequence of quads

You can frequently find large objects that contain a number of connected quads. Most graphics APIs have functions that allow you to define a sequence of quads. The vertices in the vertex list are taken as vertices of a sequence of quads that share common sides. For example, the first four vertices can define the first quad; the last two of these, together with the next two, define the next quad; and so on The order in which the vertices are presented is shown in Figure 2.4. Note the order of the vertices; instead of the expected sequence around the quads, the points in each pair have the same order. Thus the sequence 3-4 is the opposite order than would be expected, and this same sequence goes on in each additional pair of extra points. This difference is critical to note when you are implementing quad strip constructions. It might be helpful to think of this in terms of triangles, because a quad strip acts as though its vertices were specified as if it were really a triangle strip — vertices 1/2/3 followed by 2/3/4 followed by 3/4/5 etc.

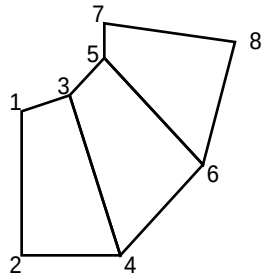


Figure 2.4: sequence of points in a quad strip

As an example of the use of quad strips and triangle fans, let's create your a model of a sphere. As we will see in the next chapter, both the GLU and GLUT toolkits include pre-built sphere models, but the sphere is a familiar object and it can be helpful to see how to create familiar things with new tools. There may also be times when you need to do things with a sphere that are difficult with the pre-built objects, so it is useful to have this example in your "bag of tricks."

In the chapter on mathematical fundamentals, we will describe the use of spherical coordinates in modeling. We can use spherical coordinates to model the sphere at first, and then we can later convert to Cartesian coordinates as we describe in that chapter to present the model to the graphics system for actual drawing. Let's think of creating a model of the sphere with N divisions around the equator and $N/2$ divisions along the prime meridian. In each case, then, the angular division will be *theta* = $360/N$ degrees. Let's also think of the sphere as having a unit radius, so it will be easier to work with later when we have transformations. Then the basic structure would be:

```
// create the two polar caps with triangle fans
doTriangleFan()      // north pole
   set vertex at (1, 0, 90)
   for i = 0 to N
      set vertex at (1, 360/i, 90-180/N)
endTriangleFan()
doTriangleFan()      // south pole
   set vertex at (1, 0, -90)
   for i = 0 to N
```

```
      set vertex at (1, 360/i, -90+180/N)
endTriangleFan()
// create the body of the sphere with quad strips
for j = -90+180/N to 90 - 180/2N
   // one quad strip per band around the sphere at a given latitude
   doQuadStrip()
      for i = 0 to 360
         set vertex at (1, i, j)
         set vertex at (1, i, j+180/N)
         set vertex at (1, i+360/N, j)
         set vertex at (1, i+360/N, j+180/N)
   endQuadStrip()
```

Note the order in which we set the points in the triangle fans and in the quad strips, as we described when we introduced these concepts; this is not immediately an obvious order and you may want to think about it a bit. Because we're working with a sphere, the quad strips as we have defined them are planar, so there is no need to divide each quad into two triangles to get planar surfaces as we might want to do for other kinds of objects.

General polygon

Some images need to include more general kinds of polygons. While these can be created by constructing them manually as collections of triangles and/or quads, it might be easier to define and display a single polygon. A graphics API will allow you to define and display a single polygon by specifying its vertices, and the vertices in the vertex list are taken as the vertices of the polygon in sequence order. As we will note in the chapter on mathematical fundamentals, many APIs can only handle *convex* polygons—polygons for which any two points in the polygon also have the entire line segment between them in the polygon. We refer you to that earlier discussion for more details.

Polyhedron

In Figure 2.1 we saw that a polyhedron is one of the basic objects we use in our modeling, especially when we will focus almost exclusively on 3D computer graphics. We specify a polyhedron by specifying all the polygons that make up its boundary. In general, most graphics APIs leave the specification of polyhedrons up to the user, which can make them fairly difficult objects to define as you are learning the subject. With experience, however, you will develop a set of polyhedra that you're familiar with and can use them with comfort.

While a graphics API may not have a general set of polyhedra, however, some provide a set of basic polyhedra that can be very useful to you. These depend on the API so we cannot be more specific here, but the next chapter includes a description of the polyhedra provided by OpenGL.

Aliasing and antialiasing

When you create a point, line, or polygon in your image, the system will define the pixels on the screen that represent the geometry within the discrete integer-coordinate 2D screen space. The standard way of selecting pixels is all-or-none: a pixel is computed to be either in the geometry, in which case it is colored as the geometry specifies, or not in the geometry, in which case it is left in whatever color it already was. Because of the relatively coarse nature of screen space, this all-or-nothing approach can leave a great deal to be desired because it created jagged edges along the space between geometry and background. This appearance is called *aliasing*, and it is shown in the left-hand image of Figure 2.4.

There are a number of techniques to reduce the effects of aliasing, and collectively the techniques are called *antialiasing*. They all work by recognizing that the boundary of a true geometry can go through individual pixels in a way that only partially covers a pixel. Each technique finds a way to account for this varying coverage and then lights the pixel according to the amount of coverage of the pixel with the geometry. Because the background may vary, this variable lighting is often managed by controlling the blending value for the pixel's color, using the color (R, G, B, A) where (R, G, B) is the geometry color and A is the proportion of the pixel covered by the object's geometry. An image that uses antialiasing is shown in the right-hand image of Figure 2.4. For more detail on color blending, see the later chapter on color.

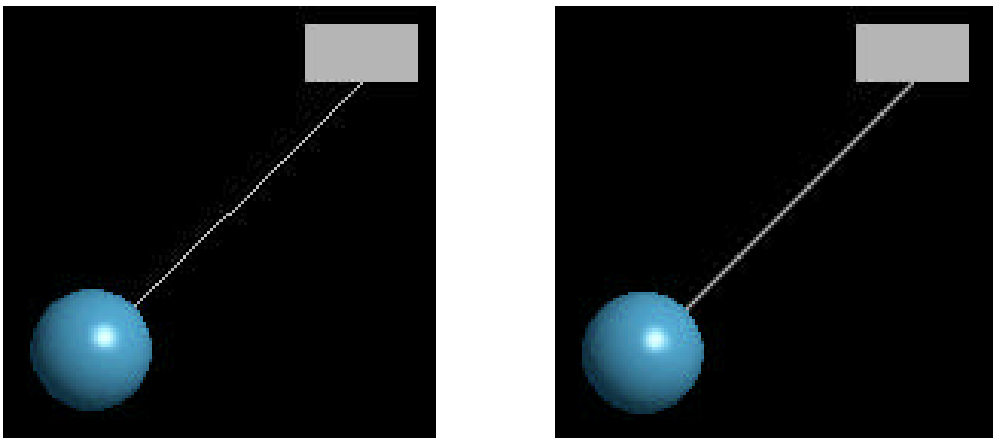


Figure 2.4: aliased lines (left) and antialiased lines (right)

As we said, there are many ways to determine the coverage of a pixel by the geometry. One way that is often used for very high-quality images is to *supersample* the pixel, that is, to assume a much higher image resolution than is really present and to see how many of these "subpixels" lie in the geometry. The proportion of subpixels that would be covered will serve as the proportion for the antialiasing value. However, supersampling is not an ordinary function of a graphics API, so we would expect a simpler approach to be used. Because APIs use linear geometries—all the basic geometry is polygon-based—it is possible to calculate exactly how the 2D world space line intersects each pixel and then how much of the pixel is covered. This is a more standard kind of API computation, though the details will certainly vary between APIs and even between different implementations of an API. You may want to look at your API's manuals for more details.

Normals

When you define the geometry of an object, you may also want or need to define the direction the object faces as well as the coordinate values for the point. This is done by defining a normal for the object. Normals are often fairly easy to obtain. In the appendix to this chapter you will see ways to calculate normals for plane polygons fairly easily; for many of the kinds of objects that are available with a graphics API, normals are built into the object definition; and if an object is defined by mathematical formulas, you can often get normals by doing some straightforward calculations.

The sphere described above is a good example of getting normals by calculation. For a sphere, the normal to the sphere at a given point is the radius vector at that point. For a unit sphere with center at the origin, the radius vector to a point has the same components as the coordinates of the point. So if you know the coordinates of the point, you know the normal at that point.

To add the normal information to the modeling definition, then, you can simply use functions that set the normal for a geometric primitive, as you would expect to have from your graphics API, and you would get code that looks something like the following excerpt from the example above:

```
for j = -90+180/M to 90-180/M  // latitude without sphere caps
   doQuadStrip()
   // one quad strip per band around the sphere at any latitude
      for i = 0 to 360          // longitude
         set normal to (1, i, j)
         set vertex at (1, i,       j      )
         set vertex at (1, i,       j+180/M)
         set vertex at (1, i+360/N, j      )
         set vertex at (1, i+360/N, j+180/M)
   endQuadStrip()
```

Data structures to hold objects

When you define a polyhedron for your graphics work, as we discussed above, there are many ways you can hold the information that describes a polyhedral graphics object. One of the simplest is the *triangle list*—an array of triples, with each set of three triples representing a separate triangle. Drawing the object is then a simple matter of reading three triples from the list and drawing the triangle. A good example of this kind of list is the STL graphics file format discussed in the chapter below on graphics hardcopy and whose formal specifications are in the Appendix.

A more effective, though a bit more complex, approach is to create three lists. The first is a *vertex list*, and it is simply an array of triples that contains all the vertices that would appear in the object. If the object is a polygon or contains polygons, the second list is an edge list that contains an entry for each edge of the polygon; the entry is an ordered pair of numbers, each of which is an index of a point in the vertex list. If the object is a polyhedron, the third is a *face list*, containing information on each of the faces in the polyhedron. Each face is indicated by listing the indices of all the edges that make up the face, in the order needed by the orientation of the face. You can then draw the face by using the indices as an indirect reference to the actual vertices. So to draw the object, you loop across the face list to draw each face; for each face you loop across the edge list to determine each edge, and for each edge you get the vertices that determine the actual geometry.

As an example, let's consider the classic cube, centered at the origin and with each side of length two. For the cube let's define the vertex array, edge array, and face array that define the cube, and let's outline how we could organize the actual drawing of the cube. We will return to this example later in this chapter and from time to time as we discuss other examples throughout the notes.

We begin by defining the data and data types for the cube. The vertices are points, which are arrays of three points, while the edges are pairs of indices of points in the point list and the faces are quadruples of indices of faces in the face list. The normals are vectors, one per face, but these are also given as arrays of three points. In C, these would be given as follows:

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];    // each face of a cube has four edges

point3 vertices[8] = {{-1.0, -1.0, -1.0},
                      {-1.0, -1.0,  1.0},
                      {-1.0,  1.0, -1.0},
                      {-1.0,  1.0,  1.0},
                      { 1.0, -1.0, -1.0},
                      { 1.0, -1.0,  1.0},
```

```
                        { 1.0,  1.0, -1.0},
                        { 1.0,  1.0,  1.0} };

point3 normals[6]   ={{ 0.0, 0.0, 1.0},
                        {-1.0, 0.0, 0.0},
                        { 0.0, 0.0,-1.0},
                        { 1.0, 0.0, 0.0},
                        { 0.0,-1.0, 0.0},
                        { 0.0, 1.0, 0.0} };

edge   edges[24]    = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                        { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                        { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                        { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                        { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                        { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

face   cube[6]      = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                        { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                        {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

Notice that in our edge list, each edge is actually listed twice—once for each direction the in which the edge can be drawn. We need this distinction to allow us to be sure our faces are oriented properly, as we will describe in the discussion on lighting and shading in later chapters. For now, we simply ensure that each face is drawn with edges in a counterclockwise direction as seen from outside that face of the cube. Drawing the cube, then, proceeds by working our way through the face list and determining the actual points that make up the cube so they may be sent to the generic (and fictitious) `setVertex(...)` and `setNormal(...)` functions. In a real application we would have to work with the details of a graphics API, but here we sketch how this would work in a pseudocode approach. In this pseudocode, we assume that there is no automatic closure of the edges of a polygon so we must list both the vertex at both the beginning and the end of the face when we define the face; if this is not needed by your API, then you may omit the first `setVertex` call in the pseudocode for the function `cube()` below.

```
void cube(void) {
   for faces 1 to 6
      start face
         setNormal(normals[i]);
         setVertex(vertices[edges[cube[face][0]][0]);
         for each edge in the face
            setVertex(vertices[edges[cube[face][edge]][1]);
      end face
}
```

We added a simple structure for a list of normals, with one normal per face, which echos the structure of the faces. This supports what is often called flat shading, or shading where each face has a single color. In many applications, though, you might want to have smooth shading, where colors blend smoothly across each face of your polygon. For this, each vertex needs to have its individual normal representing the perpendicular to the object at that vertex. In this case, you often need to specify the normal each time you specify a vertex, and a normal list that follows the vertex list would allow you to do that easily. For the code above, for example, we would not have a per-face normal but instead each `setVertex` operation could be replaced by the pair of operations

```
setNormal(normals[edges[cube[face][0]][0]);
setVertex(vertices[edges[cube[face][0]][0]);
```

Neither the simple triangle list nor the more complex structure of vertex, normal, edge, and face lists takes into account the very significant savings in memory you can get by using geometry compression techniques. There are a number of these techniques, but we only talked about line strips, triangle strips, triangle fans, and quad strips above because these are more often supported by a graphics API. Geometry compression approaches not only save space, but are also more effective for the graphics system as well because they allow the system to retain some of the information it generates in rendering one triangle or quad when it goes to generate the next one.

Additional sources of graphic objects

Interesting and complex graphic objects can be difficult to create, because it can take a lot of work to measure or calculate the detailed coordinates of each vertex needed. There are more automatic techniques being developed, including 3D scanning techniques and detailed laser rangefinding to measure careful distances and angles to points on an object that is being measured, but they are out of the reach of most college classrooms. So what do we do to get interesting objects? There are four approaches.

The first way to get models is to buy them: to go is to the commercial providers of 3D models. There is a serious market for some kinds of models, such as medical models of human structures, from the medical and legal worlds. This can be expensive, but it avoids having to develop the expertise to do professional modeling and then putting in the time to create the actual models. If you are interested, an excellent source is viewpoint.com; they can be found on the Web.

A second way to get models is to find them in places where people make them available to the public. If you have friends in some area of graphics, you can ask them about any models they know of. If you are interested in molecular models, the protein data bank (with URL `http://www.pdb.bnl.gov`) has a wide range of structure models available at no charge. If you want models of all kinds of different things, try the site `avalon.viewpoint.com`; this contains a large number of public-domain models contributed to the community by generous people over the years.

A third way to get models is to digitize them yourself with appropriate kinds of digitizing devices. There are a number of these available with their accuracy often depending on their cost, so if you need to digitize some physical objects you can compare the cost and accuracy of a number of possible kinds of equipment. The digitizing equipment will probably come with tools that capture the points and store the geometry in a standard format, which may or may not be easy to use for your particular graphics API. If it happens to be one that your API does not support, you may need to convert that format to one you use or to find a tool that does that conversion.

A fourth way to get models is to create them yourself. There are a number of tools that support high-quality interactive 3D modeling, and it is perfectly reasonable to create your models with such tools. This has the same issue as digitizing models in terms of the format of the file that the tools produce, but a good tool should be able to save the models in several formats, one of which you should be able to use fairly easily with your graphics API. It is also possible to create interesting models analytically, using mathematical approaches to generate the vertices. This is perhaps slower than getting them from other sources, but you have final control over the form and quality of the model, so perhaps it might be worth the effort. This will be discussed in the chapter on interpolation and spline modeling, for example.

If you get models from various sources, you will probably find that they come in a number of different kinds of data format. There are a large number of widely used formats for storing graphics information, and it sometimes seems as though every graphics tool uses a file format of its own. Some available tools will open models with many formats and allow you to save them in

a different format, essentially serving as format converters as well as modeling tools. In any case, you are likely to end up needing to understand some model file formats and writing your own tools to read these formats and produce the kind of internal data that you need for your models, and it may take some work to write filters that will read these formats into the kind of data structures you want for your program. Perhaps things that are "free" might cost more than things you buy if you can save the work of the conversion, but that's up to you to decide. An excellent resource on file formats is the *Encyclopedia of Graphics File Formats*, published by O'Reilly Associates, and we refer you to that book for details on particular formats.

A word to the wise

As we said above, modeling can be the most time-consuming part of creating an image, but you simply aren't going to create a useful or interesting image unless the modeling is done carefully and well. If you are concerned about the programming part of the modeling for your image, it might be best to create a simple version of your model and get the programming (or other parts that we haven't talked about yet) done for that simple version. Once you are satisfied that the programming works and that you have gotten the other parts right, you can replace the simple model—the one with just a few polygons in it—with the one that represents what you really want to present.

Transformations and Modeling

This section requires some understanding of 3D geometry, particularly a sense of how objects can be moved around in 3-space. You should also have some sense of how the general concept of stacks works.

Introduction

Transformations are probably the key point in creating significant images in any graphics system. It is extremely difficult to model everything in a scene in the place where it is to be placed, and it is even worse if you want to move things around in real time through animation and user control. Transformations let you define each object in a scene in any space that makes sense for that object, and then place it in the world space of a scene as the scene is actually viewed. Transformations can also allow you to place your eyepoint and move it around in the scene.

There are several kinds of transformations in computer graphics: projection transformations, viewing transformations, and modeling transformations. Your graphics API should support all of these, because all will be needed to create your images. Projection transformations are those that specify how your scene in 3-space is mapped to the 2D screen space, and are defined by the system when you choose perspective or orthogonal projections; viewing transformations are those that allow you to view your scene from any point in space, and are set up when you define your view environment, and modeling transformations are those you use to create the items in your scene and are set up as you define the position and relationships of those items. Together these make up the graphics pipeline that we discussed in the first chapter of these notes.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. These all maintain the basic geometry of any object to which they may be applied, and are fundamental tools to build more general models than you can create with only simple modeling techniques. Later in this chapter we will describe the relationship between objects in a scene and how you can build and maintain these relationships in your programs.

The real power of modeling transformation, though, does not come from using these simple transformations on their own. It comes from combining them to achieve complete control over your modeled objects. The individual simple transformations are combined into a composite modeling transformation that is applied to your geometry at any point where the geometry is specified. The modeling transformation can be saved at any state and later restored to that state to allow you to build up transformations that locate groups of objects consistently. As we go through the chapter we will see several examples of modeling through composite transformations.

Finally, the use of simple modeling and transformations together allows you to generate more complex graphical objects, but these objects can take significant time to display. You may want to store these objects in pre-compiled display lists that can execute much more quickly.

Definitions

In this section we outline the concept of a geometric transformation and describe the fundamental transformations used in computer graphics, and describe how these can be used to build very general graphical object models for your scenes.

Transformations

A transformation is a function that takes geometry and produces new geometry. The geometry can be anything a computer graphics systems works with—a projection, a view, a light, a direction, or

an object to be displayed. We have already talked about projections and views, so in this section we will talk about projections as modeling tools. In this case, the transformation needs to preserve the geometry of the objects we apply them to, so the basic transformations we work with are those that maintain geometry, which are the three we mentioned earlier: rotations, translations, and scaling. Below we look at each of these transformations individually and together to see how we can use transformations to create the images we need.

Our vehicle for looking at transformations will be the creation and movement of a rugby ball. This ball is basically an ellipsoid (an object that is formed by rotating an ellipse around its major axis), so it is easy to create from a sphere using scaling. Because the ellipsoid is different along one axis from its shape on the other axes, it will also be easy to see its rotations, and of course it will be easy to see it move around with translations. So we will first discuss scaling and show how it is used to create the ball, then discuss rotation and show how the ball can be rotated around one of its short axes, then discuss translations and show how the ball can be moved to any location we wish, and finally will show how the transformations can work together to create a rotating, moving ball like we might see if the ball were kicked. The ball is shown with some simple lighting and shading as described in the chapters below on these topics.

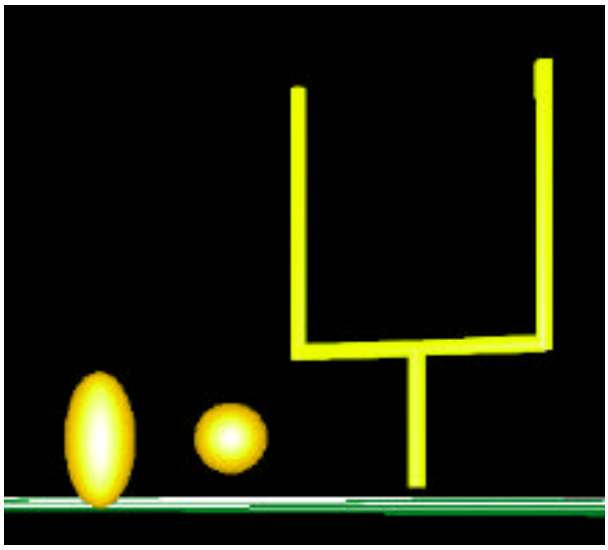


Figure 2.5: a sphere a scaled by 2.0 in the y-direction to make a rugby ball (left) and the same sphere is shown unscaled (right)

Scaling changes the entire coordinate system in space by multiplying each of the coordinates of each point by a fixed value. Each time it is applied, this changes each dimension of everything in the space. A scaling transformation requires three values, each of which controls the amount by which one of the three coordinates is changed, and a graphics API function to apply a scaling transformation will take three real values as its parameters. Thus if we have a point (x, y, z) and specify the three scaling values as Sx, Sy, and Sz, then applying the scaling transformation changes the point to $(x \quad Sx, y \quad Sy, z \cdot Sz)$. If we take a simple sphere that is centered at the origin and scale it by 2.0 in one direction (in our case, the y-coordinate or vertical direction), we get the rugby ball that is shown in Figure 2.5 next to the original sphere. It is important to note that this scaling operates on everything in the space, so if we happen to also have a unit sphere at position farther out along the axis, scaling will move the sphere farther away from the origin and will also multiply each of its coordinates by the scaling amount, possibly distorting its shape. This shows that it is most useful to apply scaling to an object defined at the origin so only the dimensions of the object will be changed.

Rotation takes everything in your space and changes each coordinate by rotating it around the origin of the geometry in which the object is defined. The rotation will always leave a line through the origin in the space fixed, that is, will not change the coordinates of any point on that line. To define a rotation transformation, you need to specify the amount of the rotation (in degrees or radians, as needed) and the line about which the rotation is done. A graphics API function to apply a rotation transformation, then, will take the angle and the line as its parameters; remember that a line through the origin can be specified by three real numbers that are the coordinates of the direction vector for that line. It is most useful to apply rotations to objects centered at the origin in order to change only the orientation with the transformation.

Translation takes everything in your space and changes each point's coordinates by adding a fixed value to each coordinate. The effect is to move everything that is defined in the space by the same amount. To define a translation transformation, you need to specify the three values that are to be added to the three coordinates of each point. A graphics API function to apply a translation, then, will take these three values as its parameters. A translation shows a very consistent treatment of everything in the space, so a translation is usually applied after any scaling or rotation in order to take an object with the right size and right orientation and place it correctly in space.

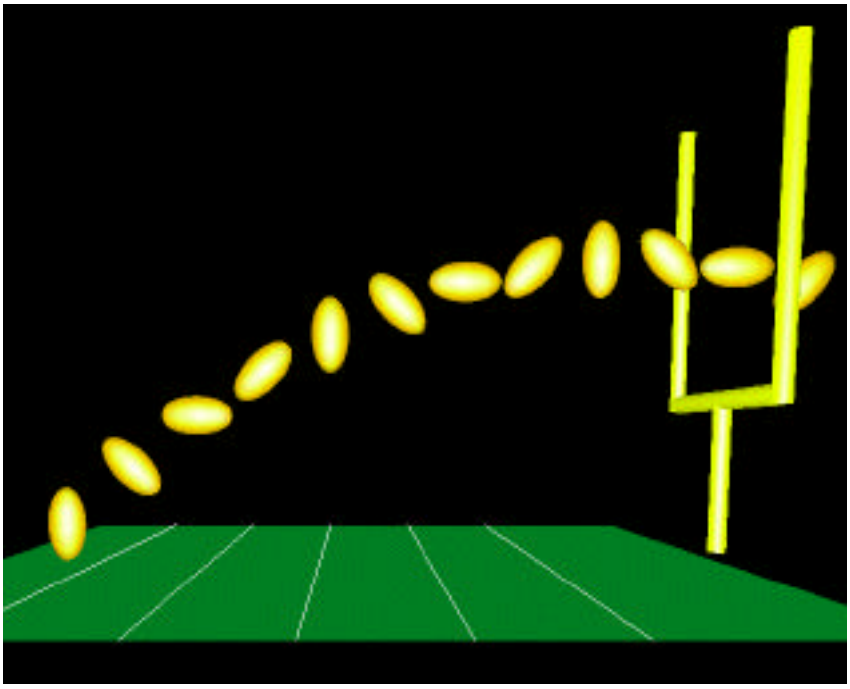


Figure 2.6: a sequence of images of the rugby ball as transformations move it through space

Finally, we put these three kinds of transformations together to create a sequence of images of the rugby ball as it moves through space, rotating as it goes, shown in Figure 2.6. This sequence was created by first defining the rugby ball with a scaling transformation and a translation putting it on the ground appropriately, creating a composite transformation as discussed in the next section. Then rotation and translation values were computed for several times in the flight of the ball, allowing us to rotate the ball by slowly-increasing amounts and placing it as if it were in a standard gravity field. Each separate image was created with a set of transformations that can be generically described by

```
translate( Tx, Ty, Tz )
rotate( angle, x-axis )
scale( 1., 2., 1. )
drawBall()
```

where the operation `drawBall()` was defined as

```
translate( Tx, Ty, Tz )
scale( 1., 2., 1. )
drawSphere()
```

Notice that the ball rotates in a slow counterclockwise direction as it travel from left to right, while the position of the ball describes a parabola as it moves, modeling the effect of gravity on the ball's flight. This kind of composite transformation constructions is described in the next section, and as we point out there, the order of these transformation calls is critical in order to achieve the effect we need.

Transformations are mathematical operations that map 3D space to 3D space, and so mathematics has standard ways to represent them. This is discussed in the next chapter, and processes such as composite transformations are linked to the standard operations on these objects.

Composite transformations

In order to achieve the image you want, you may need to apply more than one simple transformation to achieve what is called a composite transformation. For example, if you want to create a rectangular box with height *A*, width *B*, and depth *C*, with center at (*C*1, *C*2, *C*3), and oriented at an angle α relative to the *Z*-axis, you could start with a cube one unit on a side and with center at the origin, and get the box you want by applying the following sequence of operations:

first, scale the cube to the right size to create the rectangular box with dimensions *A*, *B*, and *C*,
second, rotate the cube by the angle α to the right orientation, and
third, translate the cube to the position *C*1, *C*2, *C*3.

This sequence is critical because of the way transformations work in the whole space. For example, if we rotated first and then scaled with different scale factors in each dimension, we would introduce distortions in the box. If we translated first and then rotated, the rotation would move the box to an entirely different place. Because the order is very important, we find that there are certain sequences of operations that give predictable, workable results, and the order above is the one that works best: apply scaling first, apply rotation second, and apply translation last.

The order of transformations is important in ways that go well beyond the translation and rotation example above. In general, transformations are an example of *noncommutative* operations, operations for which $f*g \neq g*f$ (that is, $f(g(x)) \neq g(f(x))$). Unless you have some experience with noncommutative operations from a course such as linear algebra, this may be a new idea. But let's look at the operations we described above: if we take the point (1, 1, 0) and apply a rotation by 90° around the *Z*-axis, we get the point (-1, 1, 0). If we then apply a translation by (2, 0, 0) we get the point (1, 1, 0) again. However, if we start with (1, 1, 0) and first apply the translation, we get (3, 1, 0) and if then apply the rotation, we get the point (-1, 3, 0) which is certainly not the same as (1, 1, 0). That is, using some pseudocode for rotations, translations, and point setting, the two code sequences

```
rotate(90, 0, 0, 1)          translate(2, 0, 0)
translate (2, 0, 0)          rotate(90, 0, 0, 1)
setPoint(1, 1, 0)            setPoint(1, 1, 0)
```

produce very different results; that is, the rotate and translate operations are not commutative.

This behavior is not limited to different kinds of transformations. Different sequences of rotations can result in different images as well. Again, if you consider the sequence of rotations in two different orders

```
rotate(60, 0, 0, 1)          rotate(90, 0, 1, 0)
rotate(90, 0, 1, 0)          rotate(60, 0, 0, 1)
scale(3, 1, .5)              scale(3, 1, .5)
cube()                       cube()
```

then the results are quite different, as is shown in Figure 2.7.

Figure 2.7: the results from two different orderings of the same rotations

Transformations are implemented as matrices for computational purposes. Recall that we are able to represent points as 4-tuples of real numbers; transformations are implemented as 4x4 matrices that map the space of 4-tuples into itself. Although we will not explicitly use this representation in our work, it is used by graphics APIs and helps explain how transformations work; for example, you can understand why transformations are not commutative by understanding that matrix multiplication is not commutative. (Try it out for yourself!) And if we realize that a 4x4 matrix is equivalent to an array of 16 real numbers, we can think of transformation stacks as stacks of such matrices. While this book does not require matrix operations for transformations, there may be times when you'll need to manipulate transformations in ways that go beyond your API, so be aware of this.

When it comes time to apply transformations to your models, we need to think about how we represent the problem for computational purposes. Mathematical notation can be applied in many ways, so your previous mathematical experience may or may not help you very much in deciding how you can think about this problem. In order to have a good model for thinking about complex transformation sequences, we will define the sequence of transformations as *last-specified, first-applied*, or in another way of thinking about it, we want to apply our functions so that the function nearest to the geometry is applied first. We can also think about this is in terms of building composite functions by multiplying the individual functions, and with the convention above we want to compose each new function by multiplying it on the right of the previous functions. So the standard operation sequence we see above would be achieved by the following algebraic sequence of operations:

```
translate * rotate * scale * geometry
```

or, thinking of multiplication as function composition, as

```
translate(rotate(scale(geometry)))
```

This might be implemented by a sequence of function calls like that below that is not intended to represent any particular API:

```
translate(C1,C2,C3); // translate to the desired point
rotate(A, Z);        // rotate by A around the Z-axis
scale(A, B, C);      // scale by the desired amounts
cube();              // define the geometry of the cube
```

At first glance, this sequence looks to be exactly the opposite of the sequence noted above. In fact, however, we readily see that the scaling operation is the function closest to the geometry (which is expressed in the function `cube()`) because of the last-specified, first-applied nature of transformations. In Figure 2.8 we see the sequence of operations as we proceed from the plain cube (at the left), to the scaled cube next, then to the scaled and rotated cube, and finally to the cube that uses all the transformations (at the right). The application is to create a long, thin, rectangular bar that is oriented at a 45° angle upwards and lies above the definition plane.

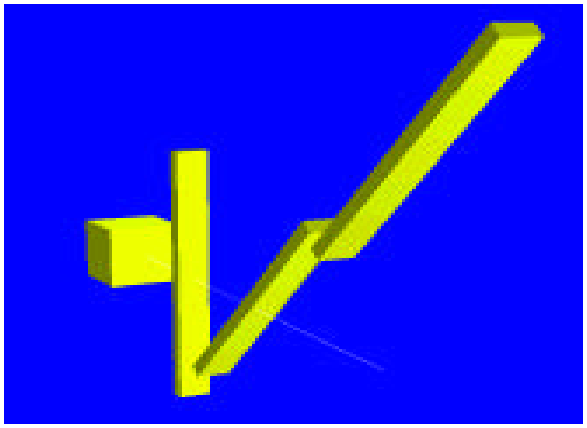


Figure 2.8: the sequence of figures as a cube is transformed

In general, the overall sequence of transformations that are applied to a model by considering the total sequence of transformations in the order in which they are specified, as well as the geometry on which they work:

```
P  V  T0 T1 T2 …  Tn Tn+1  …  Tlast  …  geometry
```

Here, `P` is the projection transformation, `V` is the viewing transformation, and `T0`, `T1`, … `Tlast` are the transformations specified in the program to model the scene, in order (`T1` is first, `Tlast` is last). The projection transformation is defined in the `reshape` function; the viewing transformation is defined in the `init` function or at the beginning of the `display` function so it is defined at the beginning of the modeling process. But the sequence is actually applied in reverse: `Tlast` is actually applied first, and `V` and finally `P` are applied last. The code would then have the definition of `P` first, the definition of `V` second, the definitions of `T0`, `T1`, … `Tlast` next in order, and the definition of the geometry last. You need to understand this sequence very well, because it's critical to understand how you build complex hierarchical models.

Transformation stacks and their manipulation

In defining a scene, we often want to define some standard pieces and then assemble them in standard ways, and then use the combined pieces to create additional parts, and go on to use these parts in additional ways. To do this, we need to create individual parts through functions that do not pay any attention to ways the parts will be used later, and then be able to assemble them into a whole. Eventually, we can see that the entire image will be a single whole that is composed of its various parts.

The key issue is that there is some kind of transformation in place when you start to define the object. When we begin to put the simple parts of a composite object in place, we will use some transformations but we need to undo the effect of those transformations when we put the next part in place. In effect, we need to save the state of the transformations when we begin to place a new part, and then to return to that transformation state (discarding any transformations we may have added past that mark) to begin to place the next part. Note that we are always adding and discarding at the end of the list; this tells us that this operation has the computational properties of a stack. We may define a stack of transformations and use it to manage this process as follows:

- as transformations are defined, they are multiplied into the current transformation in the order noted in the discussion of composite transformations above, and

- when we want to save the state of the transformation, we make a copy of the current version of the transformation and push that copy onto the stack, and apply all the subsequent transformations to the copy at the top of the stack. When we want to return to the original transformation state, we can pop the stack and throw away the copy that was removed, which gives us the original transformation so we can begin to work again at that point. Because all transformations are applied to the one at the top of the stack, when we pop the stack we return to the original context.

Designing a scene that has a large number of pieces of geometry as well as the transformations that define them can be difficult. In the next section we introduce the concept of the scene graph as a design tool to help you create complex and dynamic models both efficiently and effectively.

Compiling geometry

It can take a fair amount of time to calculate the various components of a piece of an image when that piece involves vertex lists and transformations. If an object is used frequently, and if it must be re-calculated each time it is drawn, it can make a scene quite slow to display. As a way to save time in displaying the image, many graphics APIs allow you to "compile" your geometry in a way that will allow it to be displayed much more quickly. Geometry that is to be compiled should be carefully chosen so that it is not changed between displays. If changes are needed, you will need to re-compile the object. Once you have seen what parts you can compile, you can compile them and use the compiled versions to make the display faster. We will discuss how OpenGL compiles geometry later in this chapter. If you use another API, look for details in its documentation.

A word to the wise

As we noted above, you must take a great deal of care with transformation order. It can be very difficult to look at an image that has been created with mis-ordered transformations and understand just how that erroneous example happened. In fact, there is a skill in what we might call "visual debugging"—looking at an image and seeing that it is not correct, and figuring out what errors might have caused the image as it is seen. It is important that anyone working with images become skilled in this kind of debugging. However, obviously you cannot tell than an image is wrong unless you know what a correct image should be, so you must know in general what you should be seeing. As an obvious example, if you are doing scientific images, you must know the science well enough to know when an image makes no sense.

Scene Graphs and Modeling Graphs

Introduction

In this chapter, we have defined modeling as the process of defining and organizing a set of geometry that represents a particular scene. While modern graphics APIs can provide you with a great deal of assistance in rendering your images, modeling is usually supported less well and programmers may find considerable difficulty with modeling when they begin to work in computer graphics. Organizing a scene with transformations, particularly when that scene involves hierarchies of components and when some of those components are moving, involves relatively complex concepts that need to be organized very systematically to create a successful scene. This is even more difficult when the eye point is one of the moving or hierarchically-organized parts. Hierarchical modeling has long been done by using trees or tree-like structures to organize the components of the model, and we will find this kind of approach to be very useful.

Recent graphics systems, such as Java3D and VRML 2, have formalized the concept of a *scene graph* as a powerful tool both for modeling scenes and for organizing the drawing process for those scenes. By understanding and adapting the structure of the scene graph, we can organize a careful and formal tree approach to both the design and the implementation of hierarchical models. This can give us tools to manage not only modeling the geometry of such models, but also animation and interactive control of these models and their components. In this section we will introduce the scene graph structure and will adapt it to a slightly simplified *modeling graph* that you can use to design scenes. We will also identify how this modeling graph gives us the three key transformations that go into creating a scene: the projection transformation, the viewing transformation, and the modeling transformation(s) for the scene's content. This structure is very general and lets us manage all the fundamental principles in defining a scene and translating it into a graphics API. Our terminology is based on with the scene graph of Java3D and should help anyone who uses that system understand the way scene graphs work there.

A brief summary of scene graphs

The fully-developed scene graph of the Java3D API has many different aspects and can be complex to understand fully, but we can abstract it somewhat to get an excellent model to help us think about scenes that we can use in developing the code to implement our modeling. A brief outline of the Java3D scene graph in Figure 2.9 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to beginning computer graphics. Remember that we will be simplifying some aspects of this graph before applying it to our modeling.

A *virtual universe* holds one or more (usually one) *locales*, which are essentially positions in the universe to put scene graphs. Each scene graph has two kinds of branches: *content branches*, which are to contain shapes, lights, and other content, and *view branches*, which are to contain viewing information. This division is somewhat flexible, but we will use this standard approach to build a framework to support our modeling work.

The *content branch* of the scene graph is organized as a collection of nodes that contains group nodes, transform groups, and shape nodes. A *group node* is a grouping structure that can have any number of children; besides simply organizing its children, a group can include a switch that selects which children to present in a scene. A *transform group* is a collection of modeling transformations that affect all the geometry that lies below it. The transformations will be applied to any of the transform group's children with the convention that transforms "closer" to the geometry (geometry that is defined in shape nodes lower in the graph) are applied first. A *shape node* includes both geometry and appearance data for an individual graphic unit. The geometry data includes standard 3D coordinates, normals, and texture coordinates, and can include points,

lines, triangles, and quadrilaterals, as well as triangle strips, triangle fans, and quadrilateral strips. The appearance data includes color, shading, or texture information. Lights and eye points are included in the content branch as a particular kind of geometry, having position, direction, and other appropriate parameters. Scene graphs also include shared groups, or groups that are included in more than one branch of the graph, which are groups of shapes that are included indirectly in the graph, and any change to a shared group affects all references to that group. This allows scene graphs to include the kind of template-based modeling that is common in graphics applications.

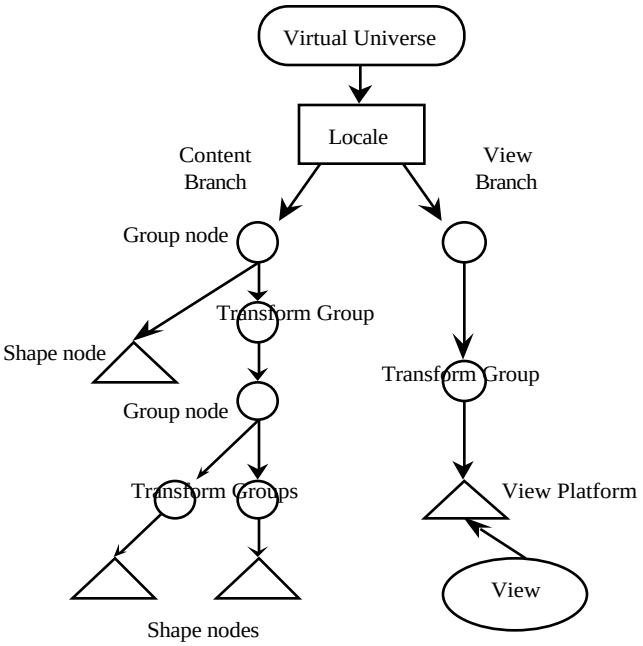


Figure 2.9: the structure of the scene graph as defined in Java3D

The *view branch* of the scene graph includes the specification of the display device, and thus the projection appropriate for that device. It also specifies the user's position and orientation in the scene and includes a wide range of abstractions of the different kinds of viewing devices that can be used by the viewer. It is intended to permit viewing the same scene on any kind of display device, including sophisticated virtual reality devices. This is a much more sophisticated approach than we need for our relatively simple modeling. We will simply consider the eye point as part of the geometry of the scene, so we set the view by including the eye point in the content branch and get the transformation information for the eye point in order to create the view transformations in the view branch.

In addition to the modeling aspect of the scene graph, Java3D also uses it to organize the processing as the scene is rendered. Because the scene graph is processed from the bottom up, the content branch is processed first, followed by the viewing transformation and then the projection transformation. However, the system does not guarantee any particular sequence in processing the node's branches, so it can optimize processing by selecting a processing order for efficiency, or can distribute the computations over a networked or multiprocessor system. Thus the Java3D programmer must be careful to make no assumptions about the state of the system when any shape node is processed. We will not ask the system to process the scene graph itself, however, because we will only use the scene graph to develop our modeling code.

An example of modeling with a scene graph

We will develop a scene graph to design the modeling for an example scene to show how this process can work. To begin, we present an already-completed scene so we can analyze how it can be created, and we will take that analysis and show how the scene graph can give us other ways to present the scene. Consider the scene as shown in Figure 2.10, where a helicopter is flying above a landscape and the scene is viewed from a fixed eye point. (The helicopter is the small green object toward the top of the scene, about 3/4 of the way across the scene toward the right.) This scene contains two principal objects: a helicopter and a ground plane. The helicopter is made up of a body and two rotors, and the ground plane is modeled as a single set of geometry with a texture map. There is some hierarchy to the scene because the helicopter is made up of smaller components, and the scene graph can help us identify this hierarchy so we can work with it in rendering the scene. In addition, the scene contains a light and an eye point, both at fixed locations. The first task in modeling such a scene is now complete: to identify all the parts of the scene, organize the parts into a hierarchical set of objects, and put this set of objects into a viewing context. We must next identify the relationship among the parts of the landscape way so we may create the tree that represents the scene. Here we note the relationship among the ground and the parts of the helicopter. Finally, we must put this information into a graph form.

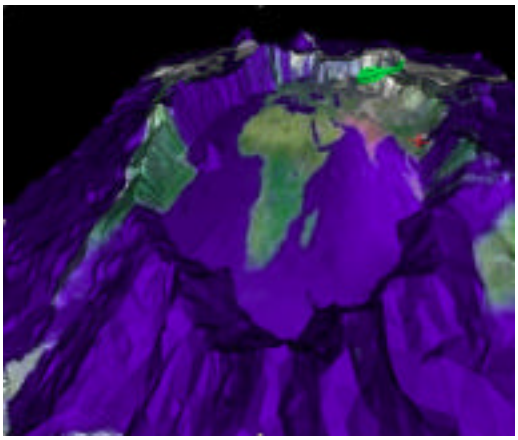


Figure 2.10: a scene that we will describe with a scene graph

The initial analysis of the scene in Figure 2.10, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 2.11. The content branch

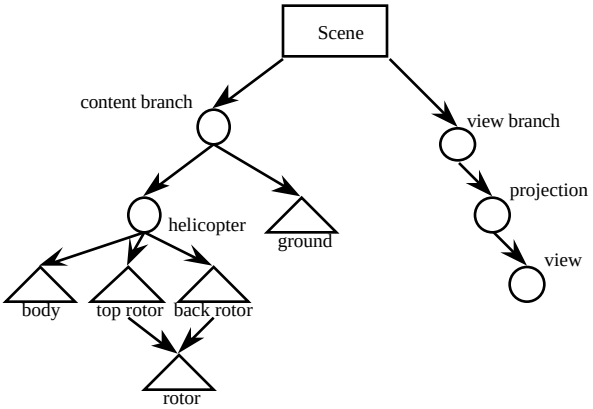


Figure 2.11: a scene graph that organizes the modeling of our simple scene

of this graph captures the organization of the components for the modeling process. This describes how content is assembled to form the image, and the hierarchical structure of this branch helps us organize our modeling components. The view branch of this graph corresponds roughly to projection and viewing. It specifies the projection to be used and develops the projection transformation, as well as the eye position and orientation to develop the viewing transformation.

This initial structure is compatible with the simple OpenGL viewing approach we discussed in the previous chapter and the modeling approach earlier in this chapter, where the view is implemented by using built-in function that sets the viewpoint, and the modeling is built from relatively simple primitives. This approach only takes us so far, however, because it does not integrate the eye into the scene graph. It can be difficult to compute the parameters of the viewing function if the eye point is embedded in the scene and moves with the other content, and later we will address that part of the question of rendering the scene.

While we may have started to define the scene graph, we are not nearly finished. The initial scene graph of Figure 2.11 is incomplete because it merely includes the parts of the scene and describes which parts are associated with what other parts. To expand this first approximation to a more complete graph, we must add several things to the graph:

- the transformation information that describes the relationship among items in a group node, to be applied separately on each branch as indicated,
- the appearance information for each shape node, indicated by the shaded portion of those nodes,
- the light and eye position, either absolute (as used in Figure 2.10 and shown Figure 2.12) or relative to other components of the model (as described later in the chapter), and
- the specification of the projection and view in the view branch.

These are all included in the expanded version of the scene graph with transformations, appearance, eyepoint, and light shown in Figure 2.12.

The content branch of this graph handles all the scene modeling and is very much like the content branch of the scene graph. It includes all the geometry nodes of the graph in Figure 2.11 as well as appearance information; includes explicit transformation nodes to place the geometry into correct sizes, positions, and orientations; includes group nodes to assemble content into logical groupings;

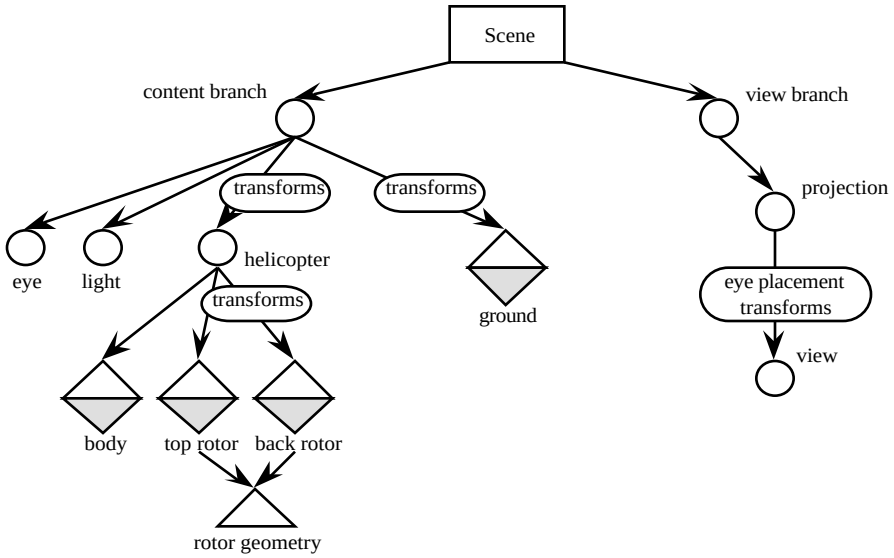


Figure 2.12: the more complete graph including transformations and appearance

and includes lights and the eye point, shown here in fixed positions. It is, of course, quite possible that in some models a light or the eye might be attached to a group instead of being positioned independently, and this can lead to some interesting examples that we describe later. In the example above, it identifies the geometry of the shape nodes such as the rotors or individual trees as shared. This might be implemented, for example, by defining the geometry of the shared shape node in a function and calling that from each of the rotor or tree nodes that uses it.

The view branch of this graph is similar to the view branch of the scene graph but is treated much more simply, containing only projection and view components. The projection component includes the definition of the projection (orthogonal or perspective) for the scene and the definition of the window and viewport for the viewing. The view component includes the information needed to create the viewing transformation, and because the eye point is placed in the content branch, this is simply a copy of the set of transformations that position the eye point in the scene as represented in the content branch.

The appearance part of the shape node is built from color, lighting, shading, texture mapping, and several other kinds of operations. Eventually each vertex of the geometry will have not only geometry, in terms of its coordinates, but also normal components, texture coordinates, and several other properties. Here, however, we are primarily concerned with the geometry content of the shape node; much of the rest of these notes is devoted to building the appearance properties of the shape node, because the appearance content is perhaps the most important part of graphics for building high-quality images.

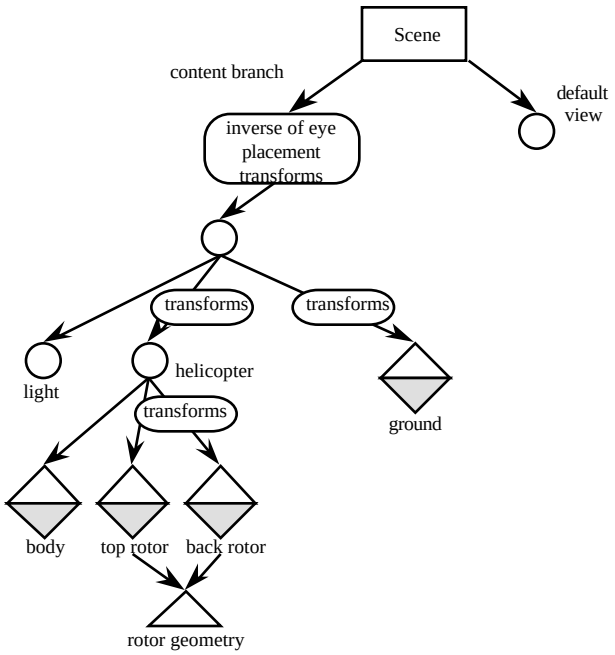


Figure 2.13: the scene graph after integrating the viewing transformation into the content branch

The scene graph for a particular image is not unique because there are many ways to organize a scene. When you have a well-defined set of transformation that place the eye point in a scene, we saw in the earlier chapter on viewing how you can take advantage of that information to organize the scene graph in a way that can define the viewing transformation explicitly and simply use the default view for the scene. As we noted there, the real effect of the viewing transformation is to be the inverse of the transformation that placed the eye. So we can explicitly compute the viewing

transformation as the inverse of the placement transformation ourselves and place that at the top of the scene graph. Thus we can restructure the scene graph of Figure 2.12 as shown in Figure 2.13 so it may take any arbitrary eye position. This will be the key point below as we discuss how to manage the eyepoint when it is a dynamic part of a scene.

It is very important to note that the scene graph need not describe a static geometry. Callbacks for user interaction and other events can affect the graph by controlling parameters of its components, as noted in the re-write guidelines in the next section. This can permit a single graph to describe an animated scene or even alternate views of the scene. The graph may thus be seen as having some components with external controllers, and the controllers are the event callback functions.

We need to extract information on the three key kinds of transformations from this graph in order to create the code that implements our modeling work. The projection transformation is straightforward and is built from the projection information in the view branch, and this is easily managed from tools in the graphics API. Because this is so straightforward, we really do not need to include it in our graph. The viewing transformation is readily created from the transformation information in the view by analyzing the eye placement transformations as we saw above, so it is straightforward to extract this and, more important, to create this transformation from the inversed of the eyepoint transformations. This is discussed in the next section of the chapter. Finally, the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn, and are discussed later in this chapter.

Because all the information we need for both the primitive geometry and all the transformations is held in this simple graph, we will call it the *modeling graph* for our scene. This modeling graph, basically a scene graph without a view branch but with the viewing information organized at the top as the inverse of the eyepoint placement transformations, will be the basis for the coding of our scenes as we describe in the remainder of the chapter.

The viewing transformation

In a scene graph with no view specified, we assume that the default view puts the eye at the origin looking in the negative z-direction with the y-axis upward. If we use a set of transformations to position the eye differently, then the viewing transformation is built by inverting those transformations to restore the eye to the default position. This inversion takes the sequence of transformations that positioned the eye and inverts the primitive transformations in reverse order, so if $T_1T_2T_3\ldots T_K$ is the original transformation sequence, the inverse is $T_K^{u}\ldots T_3^{u}T_2^{u}T_1^{u}$ where the superscript u indicates inversion, or "undo" as we might think of it.

Each of the primitive scaling, rotation, and translation transformations is easily inverted. For the scaling transformation `scale(Sx, Sy, Sz)`, we note that the three scale factors are used to multiply the values of the three coordinates when this is applied. So to invert this transformation, we must divide the values of the coordinates by the same scale factors, getting the inverse as `scale(1/Sx, 1/Sy, 1/Sz)`. Of course, this tells us quickly that the scaling function can only be inverted if none of the scaling factors are zero.

For the rotation transformation `rotate(angle, line)` that rotates space by the value `angle` around the fixed line `line`, we must simply rotate the space by the same angle in the reverse direction. Thus the inverse of the rotation transformation is `rotate(-angle, line)`.

For the translation transformation `translate(Tx, Ty, Tz)` that adds the three translation values to the three coordinates of any point, we must simply subtract those same three translation

values when we invert the transformation. Thus the inverse of the translation transformation is `translate(-Tx, -Ty, -Tz)`.

Putting this together with the information on the order of operations for the inverse of a composite transformation above, we can see that, for example, the inverse of the set of operations (written as if they were in your code)

```
translate(Tx, Ty, Tz)
rotate(angle, line)
scale(Sx, Sy, Sz)
```

is the set of operations

```
scale(1/Sx, 1/Sy, 1/Sz)
rotate(-angle, line)
translate(-Tx, -Ty, -Tz)
```

Now let us apply this process to the viewing transformation. Deriving the eye transformations from the tree is straightforward. Because we suggest that the eye be considered one of the content components of the scene, we can place the eye at any position relative to other components of the scene. When we do so, we can follow the path from the root of the content branch to the eye to obtain the sequence of transformations that lead to the eye point. That sequence of transformations is the eye transformation that we may record in the view branch.

In Figure 2.14 we show the change that results in the view of Figure 2.10 when we define the eye to be immediately behind the helicopter, and in Figure 2.15 we show the change in the scene graph of Figure 2.12 that implements the changed eye point. The eye transform consists of the transforms that places the helicopter in the scene, followed by the transforms that place the eye relative to the helicopter. Then as we noted earlier, the viewing transformation is the inverse of the eye positioning transformation, which in this case is the inverse of the transformations that placed the eye relative to the helicopter, followed by the inverse of the transformations that placed the helicopter in the scene.

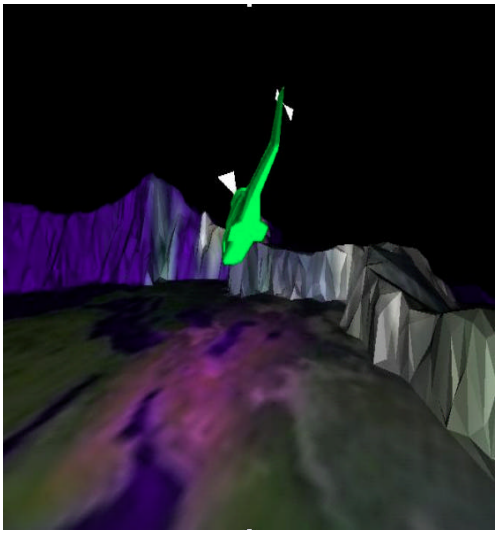


Figure 2.14: the same scene as in Figure 2.10 but with the eye point following directly behind the helicopter

This change in the position of the eye means that the set of transformations that lead to the eye point in the view branch must be changed, but the mechanism of writing the inverse of these transformations before beginning to write the definition of the scene graph still applies; only the actual transformations to be inverted will change. You might, for example, have a menu switch

that specified that the eye was to be at a fixed point or at a point following the helicopter; then the code for inverting the eye position would be a switch statement that implemented the appropriate transformations depending on the menu choice. This is how the scene graph will help you to organize the viewing process that was described in the earlier chapter on viewing.

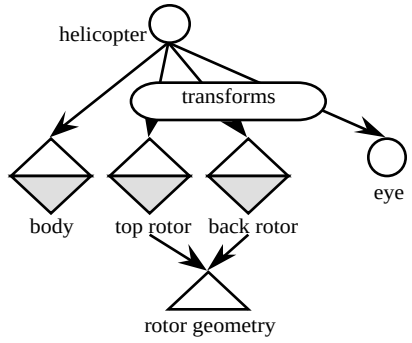


Figure 2.15: the change in the scene graph of Figure 2.10 to implement the view in Figure 2.14

With this scene graph, we can identify the set of transformations $T_a T_b T_c T_d \ldots T_i T_j T_k$ that are applied to put the helicopter in the scene, and the transformations $T_u T_v \ldots T_z$ that place the eye point relative to the helicopter. The implementation of the structure of Figure 2.13, then, is to begin the display code with the standard view, followed by $T_z^{-1} \ldots T_v^{-1} T_u^{-1}$ and then $T_k^{-1} T_j^{-1} T_i^{-1} \ldots T_d^{-1} T_c^{-1} T_b^{-1} T_a^{-1}$, before you begin to write the code for the standard scene as described in Figure 2.16 below.

The process of placing the eye point can readily be generalized. For example, if you should want to design a scene with several possible eye points and allow a user to choose among them, you can design the view branch by creating one view for each eye point and using the set of transformations leading to each eye point as the transformation for the corresponding view. You can then invert each of these sets of transformations to create the viewing transformation for each of the eye points. The choice of eye point will then create a choice of view, and the viewing transformation for that view can then be chosen to implement the user choice.

Because the viewing transformation is performed before the modeling transformations, we see from Figure 2.13 that the inverse transformations for the eye must be applied before the content branch is analyzed and its operations are placed in the code. This means that the display operation must begin with the inverse of the eye placement transformations, which has the effect of moving the eye to the top of the content branch and placing the inverse of the eye path at the front of each set of transformations for each shape node.

The scene graph and depth testing

In almost all of the images we expect to create, we would use the hidden-surface abilities provided by our graphics API. As we described in the last chapter, this will probably use some sort of depth buffer or *Z*-buffer, and the comparisons of depths for hidden surface resolution is done as the parts of the scene are drawn.

However, there may be times when you will want to avoid depth testing and take control of the sequence of drawing your scene components. One such time is described later in the chapter on color and blending, where you need to create a back-to-front drawing sequence in order to simulate transparency with blending operations. In order to do this, you will need to know the depth of each of the pieces of your scene, or the distance of that piece from the eye point. This is easy

enough to do if the scene is totally static, but when you allow pieces to move or the eye to move, it becomes much less simple.

The solution to this problem lies in doing a little extra work as you render your scene. Before you actually draw anything, but after you have updated whatever transformations you will use and whatever choices you will make to draw the current version of the scene, apply the same operations but use a tool called a *projection* that you will find with most graphics APIs. The projection operation allows you to calculate the coordinates of any point in your model space when it is transformed by the viewing and projection transformations into a point in 3D eye space. The depth of that point, then, is simply the *Z*-coordinate of the projected value. You can draw the entire scene, then, using the projection operation instead of the rendering operation, get the depth values for each piece of the scene, and use the depth values to determine the order in which you will draw the parts. The scene graph will help you make sure you have the right transformations when you project each of the parts, ensuring that you have the right depth values.

Using the modeling graph for coding

Because the modeling graph as we defined it above is intended as a learning tool and not a production tool, we will resist the temptation to formalize its definition beyond the terms we used there:

- shape node containing two components
  - geometry content
  - appearance content
- transformation node
- group node
- projection node
- view node

Because we do not want to look at any kind of automatic parsing of the modeling graph to create the scene, we will merely use the graph to help organize the structure and the relationships in the model to help you organize your code to implement your simple or hierarchical modeling. This is quite straightforward and is described in detail below.

Once you know how to organize all the components of the model in the modeling graph, you next need to write the code to implement the model. This turns out to be straightforward, and you can use a simple set of re-write guidelines that allow you to rewrite the graph as code. In this set of rules, we assume that transformations are applied in the reverse of the order they are declared, as they are in OpenGL, for example. This is consistent with your experience with tree handling in your programming courses, because you have usually discussed an expression tree which is parsed in leaf-first order. It is also consistent with the Java3D convention that transforms that are " closer" to the geometry (nested more deeply in the scene graph) are applied first.

The informal rewrite guidelines are as follows, including the rewrites for the view branch as well as the content branch:

- Nodes in the view branch involve only the window, viewport, projection, and viewing transformations. The window, viewport, and projection are handled by simple functions in the API and should be at the top of the display function.
- The viewing transformation is built from the transformations of the eye point within the content branch by copying those transformations and undoing them to place the eye effectively at the top of the content branch. This sequence should be next in the display function.
- The content branch of the modeling graph is usually maintained fully within the display function, but parts of it may be handled by other functions called from within the display, depending on the design of the scene. A function that defines the geometry of an object may be used by one or more shape nodes. The modeling may be affected by parameters set

by event callbacks, including selections of the eye point, lights, or objects to be displayed in the view.

- Group nodes are points where several elements are assembled into a single object. Each separate object is a different branch from the group node. Before writing the code for a branch that includes a transformation group, the student should push the modelview matrix; when returning from the branch, the student should pop the modelview matrix.
- Transformation nodes include the familiar translations, rotations, and scaling that are used in the normal ways, including any transformations that are part of animation or user control. In writing code from the modeling graph, students can write the transformations in the same sequence as they appear in the tree, because the bottom-up nature of the design work corresponds to the last-defined, first-used order of transformations.
- As you work your way through the modeling graph, you will need to save the state of the modeling transformation before you go down any branch of the graph from which you will need to return as the graph is traversed. Because of the simple nature of each transformation primitive, it is straightforward to undo each as needed to create the viewing transformation. This can be handled through a transformation stack that allows you to save the current transformation by pushing it onto the stack, and then restore that transformation by popping the stack.
- Shape nodes involve both geometry and appearance, and the appearance must be done first because the current appearance is applied when geometry is defined.
  - An appearance node can contain texture, color, blending, or material information that will make control how the geometry is rendered and thus how it will appear in the scene.
  - A geometry node will contain vertex information, normal information, and geometry structure information such as strip or fan organization.
- Most of the nodes in the content branch can be affected by any interaction or other event-driven activity. This can be done by defining the content by parameters that are modified by the event callbacks. These parameters can control location (by parametrizing rotations or translations), size (by parametrizing scaling), appearance (by parametrizing appearance details), or even content (by parametrizing switches in the group nodes).

We will give some examples of writing graphics code from a modeling graph in the sections below, so look for these principles as they are applied there.

In the example for Figure 2.14 above, we would use the tree to write code as shown in skeleton form in Figure 2.16. Most of the details, such as the inversion of the eye placement transformation, the parameters for the modeling transformations, and the details of the appearance of individual objects, have been omitted, but we have used indentation to show the pushing and popping of the modeling transformation stack so we can see the operations between these pairs easily. This is straightforward to understand and to organize.

Animation is simple to add to this example. The rotors can be animated by adding an extra rotation in their definition plane immediately after they are scaled and before the transformations that orient them to be placed on the helicopter body, and by updating angle of the extra rotation each time the idle event callback executes. The helicopter's behavior itself can be animated by updating the parameters of transformations that are used to position it, again with the updates coming from the idle callback. The helicopter's behavior may be controlled by the user if the positioning transformation parameters are updated by callbacks of user interaction events. So there are ample opportunities to have this graph represent a dynamic environment and to include the dynamics in creating the model from the beginning.

Other variations in this scene could by developed by changing the position of the light from its current absolute position to a position relative to the ground (by placing the light as a part of the branch group containing the ground) or to a position relative to the helicopter (by placing the light as a part of the branch group containing the helicopter). The eye point could similarly be placed

relative to another part of the scene, or either or both could be placed with transformations that are controlled by user interaction with the interaction event callbacks setting the transformation parameters.

```
display()
   set the viewport and projection as needed
   initialize modelview matrix to identity
   define viewing transformation
      invert the transformations that set the eye location
   set eye through gluLookAt with default values
   define light position         // note absolute location
   push the transformation stack       // ground
      translate
      rotate
      scale
      define ground appearance (texture)
      draw ground
   pop the transformation stack
   push the transformation stack       // helicopter
      translate
      rotate
      scale
      push the transformation stack    // top rotor
         translate
         rotate
         scale
         define top rotor appearance
         draw top rotor
      pop the transformation stack
      push the transformation stack    // back rotor
         translate
         rotate
         scale
         define back rotor appearance
         draw back rotor
      pop the transformation stack
      // assume no transformation for the body
      define body appearance
      draw body
   pop the transformation stack
   swap buffers
```

Figure 2.16: code sketch to implement the modeling in Figure 2.15

We emphasize that you should include appearance content with each shape node. Many of the appearance parameters involve a saved state in APIs such as OpenGL and so parameters set for one shape will be retained unless they are re-set for the new shape. It is possible to design your scene so that shared appearances will be generated consecutively in order to increase the efficiency of rendering the scene, but this is a specialized organization that is inconsistent with more advanced APIs such as Java3D. Thus it is very important to re-set the appearance with each shape to avoid accidentally retaining an appearance that you do not want for objects presented in later parts of your scene.

Example

We want to further emphasize the transformation behavior in writing the code for a model from the modeling graph by considering another small example. Let us consider a very simple rabbit's head as shown in Figure 2.17. This would have a large ellipsoidal head, two small spherical eyes, and two middle-sized ellipsoidal ears. So we will use the ellipsoid (actually a scaled sphere, as we saw earlier) as our basic part and will put it in various places with various orientations as needed.

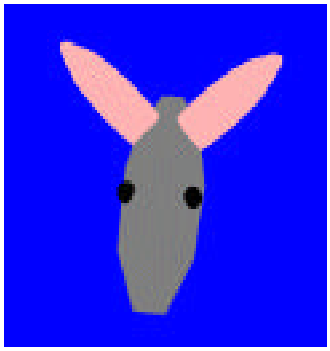


Figure 2.17: the rabbit's head

The modeling graph for the rabbit's head is shown in Figure 2.18. This figure includes all the transformations needed to assemble the various parts (eyes, ears, main part) into a unit. The fundamental geometry for all these parts is the sphere, as we suggested above. Note that the transformations for the left and right ears include rotations; these can easily be designed to use a parameter for the angle of the rotation so that you could make the rabbit's ears wiggle back and forth.

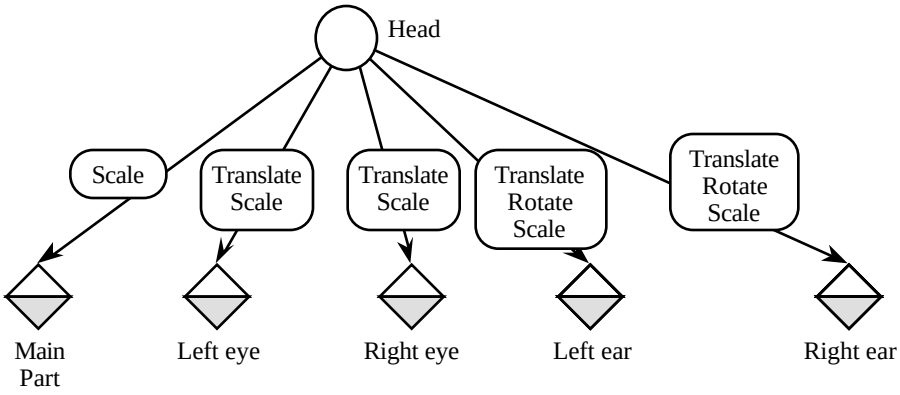


Figure 2.18: the modeling graph for the rabbit's head

To write the code to implement the modeling graph for the rabbit's head, then, we would apply the following sequence of actions on the modeling transformation stack:

- push the modeling transformation stack
- apply the transformations to create the head, and define the head:
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left eye relative to the head, and define the eye:
  translate

scale
draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right eye relative to the head, and define the eye:
  translate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack

You should trace this sequence of operations carefully and watch how the head is drawn. Note that if you were to want to put the rabbit's head on a body, you could treat this whole set of operations as a single function `rabbitHead()` that is called between operations push and pop the transformation stack, with the code to place the head and move it around lying above the function call. This is the fundamental principle of hierarchical modeling — to create objects that are built of other objects, finally reducing the model to simple geometry at the lowest level. In the case of the modeling graph, that lowest level is the leaves of the tree, in the shape nodes.

The transformation stack we have used informally above is a very important consideration in using a scene graph structure. It may be provided by your graphics API or it may be something you need to create yourself; even if it provided by the API, there may be limits on the depth of the stack that will be inadequate for some projects and you may need to create your own. We will discuss this in terms of the OpenGL API later in this chapter.

Using standard objects to create more complex scenes

The example of transformation stacks is, in fact, a larger example—an example of using standard objects to define a larger object. In a program that defined a scene that needed rabbits, we would create the rabbit head with a function `rabbitHead()` that has the content of the code we used (and that is given below) and would apply whatever transformations would be needed to place a rabbit head properly on each rabbit body. The rabbits themselves could be part of a larger scene, and you could proceed in this way to create however complex a scene as you wish.

Questions

1. We know that we can model any polyhedron with triangles, but why can you model a sphere with triangle fans for the polar caps and quad strips for the rest of the object?

2. Put yourself in a familiar environment, but imagine the environment simplified so that it is made up of only boxes, cylinders, and other very basic shapes. Imagine further that your

environment has only one door and that everything in the room has to come in that door. Write out the sequence of transformations that would have to be done to place everything in its place in your environment. Now imagine that each of these basic shapes starts out as a standard shape: a unit cube, a cylinder with diameter one and height one, and the like; write out the sequence of transformations that would have to be done to make each object from these basic objects. Finally, if the door would only admit basic objects, put together these two processes to write out the full transformations to create the objects and place them in the space.

3. Now take the environment above and write a scene graph that describes the whole scene, using the basic shapes and transformations you identified in the previous question. Also place your eye in the scene graph starting with a standard view of you standing in the doorway and facing directly into the room. Now imagine that on a table in the space there is a figure of a ballerina spinning around and around, and identify the way the transformations in the scene graph would handle this moving object.

Exercises

4. Calculate the coordinates of the vertices of the simpler regular polyhedra: the cube, the tetrahedron, and the octagon. For the octagon and tetrahedron, try using spherical coordinates and converting them to rectangular coordinates; see the chapter on mathematics for modeling.

5. Verify that for any x, y, z, and w, the point $(x/w,\ y/w,\ z/w,\ 1)$ is the intersection of the line segment from $(x,\ y,\ z,\ w)$ to $(0,\ 0,\ 0,\ 0)$, and the hyperplane $\{\ (a,\ b,\ c,\ 1)\ |$ arbitrary $a,\ b,\ c\ \}$.

6. Show how you can define a cube as six quads. Show how you can refine that definition to write a cube as two quad strips. Can you write a cube as one quad strip?

7. Show how you can write any polygon, convex or not, as a set of triangles. Show further how you can write any *convex* polygon as a triangle fan.

8. Define a polygon in 2D space that is reasonably large and having a side that is not parallel to one of the axes. Find a unit square in the 2D space that intersects that side, and calculate the proportion of the polygon that lies within the unit square. If the square represents a pixel, draw conclusions about what proportion of the pixel's color should come from the polygon and what proportion from the background.

9. The code for the normals to a quad on page 2.9 is not accurate because it uses the normal at a vertex instead of the normal in the middle of the quad. How should you calculate the normal so that it is the face normal and not a vertex normal?

10. Make a basic object with no symmetries, and apply simple rotation, simple translation, and simple scaling to it; see what you get. Then apply second and third transformations after you have applied the first transformation and again see what you get. Show why the order of the transformations matters by applying the same transformations in different order and seeing the results.

11. Scene graphs are basically trees, though different branches may share common shape objects. As trees, they can be traversed in any way that is convenient. Show how you might choose the way you would traverse a scene graph in order to draw back-to-front if you knew the depth of each object in the tree.

12. Add a mouth and tongue to the rabbit's head, and modify the scene graph for the rabbit's head to have the rabbit stick out its tongue and wiggle it around.

13. Define a scene graph for a carousel, or merry-go-round. This object has a circular disk as its floor, a cone as its roof, a collection of posts that connect the floor to the roof, and a collection of animals in a circle just inside the outside diameter of the floor, each parallel to the tangent to the floor at the point on the edge nearest the animal. The animals will go up and down in a periodic way as the carousel goes around. You may assume that each animal is a primitive and not try to model it, but you should carefully define all the transformations that build the carousel and place the animals.

Experiments

14. Get some of the models from the `avalon.viewpoint.com` site and examine the model file to see how you could present the model as a sequence of triangles or other graphics primitives.

15. Write the code for the scene graph of the familiar space from question 3, including the code that manages the inverse transformations for the eye point. Now identify a simple path for the eye, created by parametrizing some of the transformations that place the eye, and create an animation of the scene as it would be seen from the moving eye point.

Chapter 3: Implementing Modeling in OpenGL

This chapter discusses the way OpenGL implements the general modeling discussion of the last chapter. This includes specifying geometry, specifying points for that geometry in model space, specifying normals for these vertices, and specifying and managing transformations that move these objects from model space into the world coordinate system. It also includes the set of operations that implement polygons, including those that provide the geometry compression that was described in the previous chapter. Finally, it discusses some pre-built models that are provided by the OpenGL and GLUT environments to help you create your scenes more easily.

The OpenGL model for specifying geometry

In defining your model for your program, you will use a single function to specify the geometry of your model to OpenGL. This function specifies that geometry is to follow, and its parameter defines the way in which that geometry is to be interpreted for display:

```
glBegin(mode);
// vertex list: point data to create a primitive object in
// the drawing mode you have indicated
// normals may also be specified here
glEnd();
```

The vertex list is interpreted as needed for each drawing mode, and both the drawing modes and the interpretation of the vertex list are described in the discussions below. This pattern of `glBegin(mode) - vertex list - glEnd` uses different values of the `mode` to establish the way the vertex list is used in creating the image. Because you may use a number of different kinds of components in an image, you may use this pattern several times for different kinds of drawing. We will see a number of examples of this pattern in this module.

In OpenGL, point (or vertex) information is presented to the computer through a set of functions that go under the general name of `glVertex*(…)`. These functions enter the numeric value of the vertex coordinates into the OpenGL pipeline for the processing to convert them into image information. We say that `glVertex*(…)` is a *set* of functions because there are many functions that differ only in the way they define their vertex coordinate data. You may want or need to specify your coordinate data in any standard numeric type, and these functions allow the system to respond to your needs.

- If you want to specify your vertex data as three separate real numbers, or floats (we'll use the variable names `x`, `y`, and `z`, though they could also be float constants), you can use `glVertex3f(x,y,z)`. Here the character `f` in the name indicates that the arguments are floating-point; we will see below that other kinds of data formats may also be specified for vertices.
- If you want to define your coordinate data in an array, you could declare your data in a form such as `glFloat x[3]` and then use `glVertex3fv(x)` to specify the vertex. Adding the letter `v` to the function name specifies that the data is in vector form (actually a pointer to the memory that contains the data, but an array's name is really such a pointer). Other dimensions besides 3 are also possible, as noted below.

Additional versions of the functions allow you to specify the coordinates of your point in two dimensions (`glVertex2*`); in three dimensions specified as integers (`glVertex3i`), doubles (`glVertex3d`), or shorts (`glVertex3s`); or as four-dimensional points (`glVertex4*`). The four-dimensional version uses homogeneous coordinates, as described earlier in this chapter. You will see some of these used in the code examples later in this chapter.

One of the most important things to realize about modeling in OpenGL is that you can call your own functions between a `glBegin(mode)` and `glEnd()` pair to determine vertices for your vertex list. Any vertices these functions define by making a `glVertex*(…)` function call will be

added to the vertex list for this drawing mode. This allows you to do whatever computation you need to calculate vertex coordinates instead of creating them by hand, saving yourself significant effort and possibly allowing you to create images that you could not generate by hand. For example, you may include various kind of loops to calculate a sequence of vertices, or you may include logic to decide which vertices to generate. An example of this way to generate vertices is given among the first of the code examples toward the end of this module.

Another important point about modeling is that a great deal of other information can go between a `glBegin(mode)` and `glEnd()` pair. We will see the importance of including information about vertex normals in the chapters on lighting and shading, and of including information on texture coordinates in the chapter on texture mapping. So this simple construct can be used to do much more than just specify vertices. Although you may carry out whatever processing you need within the `glBegin(mode)` and `glEnd()` pair, there are a limited number of OpenGL operations that are permitted here. In general, the available OpenGL operations here are `glVertex`, `glColor`, `glNormal`, `glTexCoord`, `glEvalCoord`, `glEvalPoint`, `glMaterial`, `glCallList`, and `glCallLists`, although this is not a complete list. Your OpenGL manual will give you additional information if needed.

Point and points mode

The mode for drawing points with the `glBegin` function is named `GL_POINTS`, and any vertex data between `glBegin` and `glEnd` is interpreted as the coordinates of a point we wish to draw. If we want to draw only one point, we provide only one vertex between `glBegin` and `glEnd`; if we want to draw more points, we provide more vertices between them. If you use points and want to make each point more visible, the function `glPointSize(float size)` allows you to set the size of each point, where `size` is any nonnegative real value and the default size is 1.0.

The code below draws a sequence of points in a straight line. This code takes advantage of fact that we can use ordinary programming processes to define our models, showing we need not hand-calculate points when we can determine them by an algorithmic approach. We specify the vertices of a point through a function `pointAt()` that calculates the coordinates and calls the `glVertex*()` function itself, and then we call that function within the `glBegin/glEnd` pair. The function calculates points on a spiral along the z-axis with x- and y-coordinates determined by functions of the parameter `t` that drives the entire spiral.

```
void pointAt(int i) {
   glVertex3f(fx(t)*cos(g(t)),fy(t)*sin(g(t)),0.2*(float)(5-i));
}

void pointSet( void ) {
   int i;

   glBegin(GL_POINTS);
      for ( i=0; i<10; i++ )
      pointAt(i);
   glEnd();
}
```

Some functions that drive the x- and y-coordinates may be familiar to you through studies of functions of polar coordinates in previous mathematics classes, and you are encouraged to try out some possibilities on your own.

Line segments

To draw line segments, we use the `GL_LINES` mode for `glBegin/glEnd`. For each segment we wish to draw, we define the vertices for the two endpoints of the segment. Thus between `glBegin` and `glEnd` each pair of vertices in the vertex list defines a separate line segment.

Line strips

Connected lines are called *line strips* in OpenGL, and you can specify them by using the mode `GL_LINE_STRIP` for `glBegin/glEnd`. The vertex list defines the line segments as noted in the general discussion of connected lines above, so if you have N vertices, you will have N-1 line segments. With either line segments or connected lines, we can set the line width to emphasize (or de-emphasize) a line. Heavier line widths tend to attract more attention and give more emphasis than lighter line widths. The line width is set with the `glLineWidth(float width)` function. The default value of `width` is 1.0 but any nonnegative width can be used.

As an example of a line strip, let's consider a parametric curve. Such curves in 3-space are often interesting objects of study. The code below define a rough spiral in 3-space that is a good (though simple) example of using a single parameter to define points on a parametric curve so it can be drawn for study.

```
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=10; i++ )
      glVertex3f(2.0*cos(3.14159*(float)i/5.0),
         2.0*sin(3.14159*(float)i/5.0),0.5*(float)(i-5));
glEnd();
```

This can be made much more sophisticated by increasing the number of line segments, and the code can be cleaned up a bit as described in the code fragment below. Simple experiments with the `step` and `zstep` variables will let you create other versions of the spiral as experiments.

```
#define PI 3.14159
#define N 100
step = 2.0*PI/(float)N;
zstep = 2.0/(float)N;
glBegin(GL_LINE_STRIP);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),2.0*cos(step*(float)i),
      -1.0+zstep*(float)i);
glEnd();
```

If this spiral is presented in a program that includes some simple rotations, you can see the spiral from many points in 3-space. Among the things you will be able to see are the simple sine and cosine curves, as well as one period of the generic shifted sine curve.

Line loops

A line loop is just like a line strip except that an additional line segment is drawn from the last vertex in the list to the first vertex in the list, creating a closed loop. There is little more to be said about line loops; they are specified by using the mode `GL_LINE_LOOP`.

Triangle

To draw unconnected triangles, you use `glBegin/glEnd` with the mode `GL_TRIANGLES`. This is treated exactly as discussed in the previous chapter and produces a collection of triangles, one for each three vertices specified.

Sequence of triangles

OpenGL provides both of the standard geometry-compression techniques to assemble sequences of triangles: triangle strips and triangle fans. Each has its own mode for `glBegin/glEnd`: `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN` respectively. These behave exactly as described in the general section above.

Because there are two different modes for drawing sequences of triangles, we'll consider two examples in this section. The first is a triangle fan, used to define an object whose vertices can be seen as radiating from a central point. An example of this might be the top and bottom of a sphere, where a triangle fan can be created whose first point is the north or south pole of the sphere. The second is a triangle strip, which is often used to define very general kinds of surfaces, because most surfaces seem to have the kind of curvature that keeps rectangles of points on the surface from being planar. In this case, triangle strips are much better than quad strips as a basis for creating curved surfaces that will show their surface properties when lighted.

The triangle fan (that defines a cone, in this case) is organized with its vertex at point `(0.0,1.0,0.0)` and with a circular base of radius 0.5 in the XZ-plane. Thus the cone is oriented towards the y-direction and is centered on the y-axis. This provides a surface with unit diameter and height, as shown in Figure 3.1. When the cone is used in creating a scene, it can easily be defined to have whatever size, orientation, and location you need by applying appropriate modeling transformations in an appropriate sequence. Here we have also added normals and flat shading to emphasize the geometry of the triangle fan, although the code does not reflect this.

```
glBegin(GL_TRIANGLE_FAN);
   glVertex3f(0., 1.0, 0.);  // the point of the cone
   for (i=0; i < numStrips; i++) {
      angle = 2. * (float)i * PI / (float)numStrips;
      glVertex3f(0.5*cos(angle), 0.0, 0.5*sin(angle));
      // code to calculate normals would go here
   }
glEnd();
```

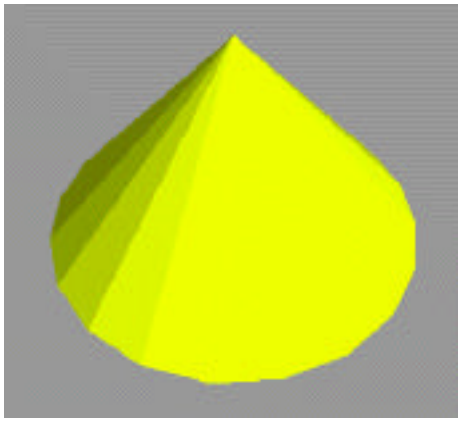


Figure 3.1: the cone produced by the triangle fan

The triangle strip example is based on an example of a function surface defined on a grid. Here we describe a function whose domain is in the X-Z plane and whose values are shown as the Y-value of each vertex. The grid points in the X-Z domain are given by functions `XX(i)` and `ZZ(j)`, and

the values of the function are held in an array, with `vertices[i][j]` giving the value of the function at the grid point `(XX(i),ZZ(j))` as defined in the short example code fragment below.

```
for ( i=0; i<XSIZE; i++ )
   for ( j=0; j<ZSIZE; j++ ) {
      x = XX(i);
      z = ZZ(j);
      vertices[i][j] = (x*x+2.0*z*z)/exp(x*x+2.0*z*z+t);
   }
```

The surface rendering can then be organized as a nested loop, where each iteration of the loop draws a triangle strip that presents one section of the surface. Each section is one unit in the X-direction that extends across the domain in the Z-direction. The code for such a strip is shown below, and the resulting surface is shown in Figure 3.2. Again, the code that calculates the normals is omitted; this example is discussed further and the normals are developed in the later chapter on shading. This kind of surface is explored in more detail in the chapters on scientific applications of graphics.

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      glBegin(GL_TRIANGLE_STRIP);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
      glEnd();
   }
```

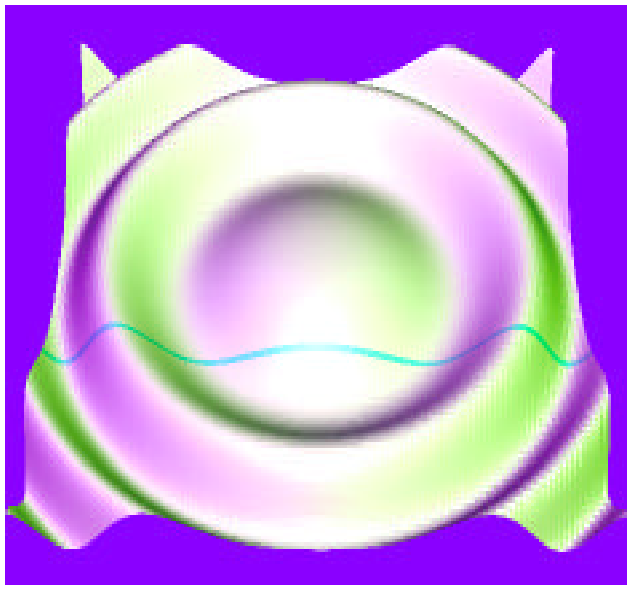


Figure 3.2: the full surface created by triangle strips, with a single strip highlighted in cyan

This example is a white surface lighted by three lights of different colors, a technique we describe in the chapter on lighting. This surface example is also briefly revisited in the quads discussion below. Note that the sequence of points here is slightly different here than it is in the example below because of the way quads are specified. In this example instead of one quad, we will have two triangles—and if you rework the example below to use quad strips instead of simple quads to

display the mathematical surface, it is simple to make the change noted here and do the surface with extended triangle strips.

Quads

To create a set of one or more distinct quads you use `glBegin/glEnd` with the `GL_QUADS` mode. As described earlier, this will take four vertices for each quad. An example of an object based on quadrilaterals would be the function surface discussed in the triangle strip above. For quads, the code for the surface looks like this:

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ ) {
      // quad sequence: points (i,j),(i+1,j),(i+1,j+1),(i,j+1)
      glBegin(GL_QUADS);
         glVertex3f(XX(i),vertices[i][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
         glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
         glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
      glEnd();
   }
```

Note that neither this surface nor the one composed from triangles is going to look very good yet because it does not yet contain any lighting or color information. These will be added in later chapters as this concept of function surfaces is re-visited when we discuss lighting and color.

Quad strips

To create a sequence of quads, the mode for `glBegin/glEnd` is `GL_QUAD_STRIP`. This operates in the way we described at the beginning of the chapter, and as we noted there, the order in which the vertices are presented is different from that in the GL_QUADS mode. Be careful of this when you define your geometry or you may get a very unusual kind of display!

In a fairly common application, we can create long, narrow tubes with square cross-section. This can be used as the basis for drawing 3-D coordinate axes or for any other application where you might want to have, say, a beam in a structure. The quad strip defined below creates the tube oriented along the Z-axis with the cross-section centered on that axis. The dimensions given make a unit tube—a tube that is one unit in each dimension, making it actually a cube. These dimensions will make it easy to scale to fit any particular use.

```
#define RAD 0.5
#define LEN 1.0
glBegin(GL_QUAD_STRIP);
   glVertex3f( RAD, RAD, LEN ); // start of first side
   glVertex3f( RAD, RAD, 0.0 );
   glVertex3f(-RAD, RAD, LEN );
   glVertex3f(-RAD, RAD, 0.0 );
   glVertex3f(-RAD,-RAD, LEN ); // start of second side
   glVertex3f(-RAD,-RAD, 0.0 );
   glVertex3f( RAD,-RAD, LEN ); // start of third side
   glVertex3f( RAD,-RAD, 0.0 );
   glVertex3f( RAD, RAD, LEN ); // start of fourth side
   glVertex3f( RAD, RAD, 0.0 );
glEnd();
```

You can also get the same object by using the GLUT cube that is discussed below and applying appropriate transformations to center it on the Z-axis.

General polygon

The GL_POLYGON mode for glBegin/glEnd is used to allow you to display a single convex polygon. The vertices in the vertex list are taken as the vertices of the polygon in sequence order, and we remind you that the polygon needs to be convex. It is not possible to display more than one polygon with this operation because the function will always assume that whatever points it receives go in the same polygon.

Probably the simplest kind of multi-sided convex polygon is the regular N-gon, an N-sided figure with all edges of equal length and all interior angles between edges of equal size. This is simply created (in this case, for N=7), again using trigonometric functions to determine the vertices.

```
#define PI 3.14159
#define N 7
step = 2.0*PI/(float)N;
glBegin(GL_POLYGON);
   for ( i=0; i<=N; i++)
      glVertex3f(2.0*sin(step*(float)i),
                 2.0*cos(step*(float)i),0.0);
glEnd();
```

Note that this polygon lives in the XY-plane; all the Z-values are zero. This polygon is also in the default color (white) because we have not specified the color to be anything else. This is an example of a "canonical" object—an object defined not primarily for its own sake, but as a template that can be used as the basis of building another object as noted later, when transformations and object color are available. An interesting application of regular polygons is to create regular polyhedra—closed solids whose faces are all regular N-gons. These polyhedra are created by writing a function to draw a simple N-gon and then using transformations to place these properly in 3-space to be the boundaries of the polyhedron.

Antialiasing

As we saw in the previous chapter, geometry drawn with antialiasing is smoother and less "jaggy" than geometry drawn in the usual "all-or-nothing" pixel mode. OpenGL provides some capabilities for antialiasing by allowing you to enable point smoothing, line smoothing, and/or polygon smoothing. These are straightforward to specify, but they operate with color blending and there may be some issues around the order in which you draw your geometry. OpenGL calculates the coverage factor for antialiasing based on computing the proportion of a pixel that is covered by the geometry being presented, as described in the previous chapter. For more on RGBA blending and the order in which drawing is done, see the later chapter on color and blending.

To use the built-in OpenGL antialiasing, choose the various kinds of point, line, or polygon smoothing with the glEnable(...) function. Each implementation of OpenGL will define a default behavior for smoothing, so you may want to override that default by defining your choice with the glHint(...) function. The appropriate pairs of enable/hint are shown here:

```
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POLYGON_SMOOTH);
glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);
```

There is a more sophisticated kind of polygon smoothing involving entire scene antialiasing, done by drawing the scene into the accumulation buffer with slight offsets so that boundary pixels will be chosen differently for each version. This is a time-consuming process and is generally considered a more advanced use of OpenGL than we are assuming in this book. We do discuss the accumulation buffer in a later chapter when we discuss motion blur, but we will not go into more detail here.

The cube we will use in many examples

Because a cube is made up of six square faces, it is very tempting to try to make the cube from a single quad strip. Looking at the geometry, though, it is impossible to make a single quad strip go around the cube; in fact, the largest quad strip you can create from a cube's faces has only four quads. It is possible to create two quad strips of three faces each for the cube (think of how a baseball is stitched together), but here we will only use a set of six quads whose vertices are the eight vertex points of the cube. Below we repeat the declarations of the vertices, normals, edges, and faces of the cube from the previous chapter. We will use the `glVertex3fv(…)` vertex specification function within the specification of the quads for the faces.

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];  // each face of a cube has four edges

   point3 vertices[8] = {{-1.0, -1.0, -1.0},
                         {-1.0, -1.0,  1.0},
                         {-1.0,  1.0, -1.0},
                         {-1.0,  1.0,  1.0},
                         { 1.0, -1.0, -1.0},
                         { 1.0, -1.0,  1.0},
                         { 1.0,  1.0, -1.0},
                         { 1.0,  1.0,  1.0} };

   point3 normals[6]  = {{ 0.0, 0.0, 1.0},
                         {-1.0, 0.0, 0.0},
                         { 0.0, 0.0,-1.0},
                         { 1.0, 0.0, 0.0},
                         { 0.0,-1.0, 0.0},
                         { 0.0, 1.0, 0.0} };

   edge   edges[24]   = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                         { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                         { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                         { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                         { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                         { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

   face   cube[6]     = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                         { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                         {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

As we said before, drawing the cube proceeds by working our way through the face list and determining the actual points that make up the cube. We will expand the function we gave earlier to write the actual OpenGL code below. Each face is presented individually in a loop within the `glBegin-glEnd` pair, and with each face we include the normal for that face. Note that only the first vertex of the first edge of each face is identified, because the GL_QUADS drawing mode

takes each set of four vertices as the vertices of a quad; it is not necessary to close the quad by including the first point twice.

```
void cube(void) {
   int face, edge;
   glBegin(GL_QUADS);
      for (face = 0; face < 6; face++) {
         glNormal3fv(normals[face];
         for (edge = 0; edge < 4; edge++)
            glVertex3fv(vertices[edges[cube[face][edge]][0]]);
      }
   glEnd();
}
```

This cube is shown in Figure 3.3, presented through the six steps of adding individual faces (the faces are colored in the typical RGBCMY sequence so you may see each added in turn). This approach to defining the geometry is actually a fairly elegant way to define a cube, and takes very little coding to carry out. However, this is not the only approach we could take to defining a cube. Because the cube is a regular polyhedron with six faces that are squares, it is possible to define the cube by defining a standard square and then using transformations to create the faces from this master square. Carrying this out is left as an exercise for the student.

This approach to modeling an object includes the important feature of specifying the normals (the vectors perpendicular to each face) for the object. We will see in the chapters on lighting and shading that in order to get the added realism of lighting on an object, we must provide information on the object's normals, and it was straightforward to define an array that contains a normal for each face. Another approach would be to provide an array that contains a normal for each vertex if you would want smooth shading for your model; see the chapter on shading for more details. We will not pursue these ideas here, but you should be thinking about them when you consider modeling issues with lighting.

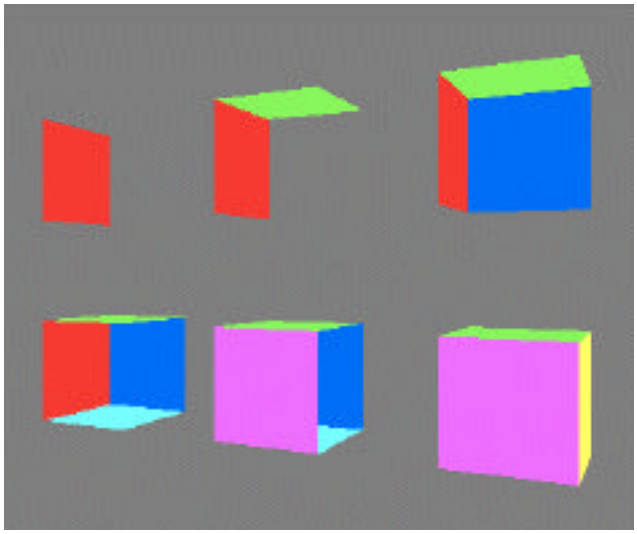


Figure 3.3: the cube as a sequence of quads

Additional objects with the OpenGL toolkits

Modeling with polygons alone would require you to write many standard graphics elements that are so common, any reasonable graphics system should include them. OpenGL includes the

OpenGL Utility Library, GLU, with many useful functions, and most releases of OpenGL also include the OpenGL Utility Toolkit, GLUT. We saw in the first chapter that GLUT includes window management functions, and both GLU and GLUT include a number of built-in graphical elements that you can use. This chapter describes a number of these elements.

The objects that these toolkits provide are defined with several parameters that define the details, such as the resolution in each dimension of the object with which the object is to be presented. Many of these details are specific to the particular object and will be described in more detail when we describe each of these.

GLU quadric objects

The GLU toolkit provides several general quadric objects, which are objects defined by quadric equations (polynomial equations in three variables with degree no higher than two in any term), including Spheres (`gluSphere`), cylinders (`gluCylinder`), and disks (`gluDisk`). Each GLU primitive is declared as a GLUquadric and is allocated with the function

```
GLUquadric* gluNewQuadric( void )
```

Each quadric object is a surface of revolution around the z-axis. Each is modeled in terms of subdivisions around the z-axis, called slices, and subdivisions along the z-axis, called stacks. Figure 3.4 shows an example of a typical pre-built quadric object, a GLUT wireframe sphere, modeled with a small number of slices and stacks so you can see the basis of this definition.

The GLU quadrics are very useful in many modeling circumstances because you can use scaling and other transformations to create many common objects from them. The GLU quadrics are also useful because they have capabilities that support many of the OpenGL rendering capabilities that support creating interesting images. You can determine the drawing style with the `gluQuadricDrawStyle()` function that lets you select whether you want the object filled, wireframe, silhouette, or drawn as points. You can get normal vectors to the surface for lighting models and smooth shading with the `gluQuadricNormals()` function that lets you choose whether you want no normals, or normals for flat or smooth shading. Finally, with the `gluQuadricTexture()` function you can specify whether you want to apply texture maps to the GLU quadrics in order to create objects with visual interest. See later chapters on lighting and on texture mapping for the details.

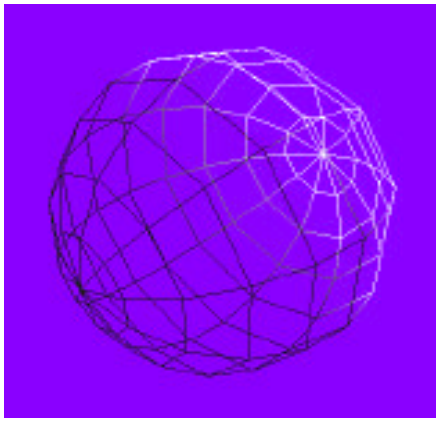


Figure 3.4: A GLUT wireframe sphere with 10 slices and 10 stacks

Below we describe each of the GLU primitives by listing its function prototype; more details may be found in the GLU section of your OpenGL manual.

GLU cylinder:

```
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top,
GLdouble height, GLint slices, GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
base is the radius of the cylinder at z = 0, the base of the cylinder
top is the radius of the cylinder at z = height, and
height is the height of the cylinder.

GLU disk:

The GLU disk is different from the other GLU primitives because it is only two-dimensional, lying entirely within the X-Y plane. Thus instead of being defined in terms of stacks, the second granularity parameter is loops, the number of concentric rings that define the disk.

```
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer,
GLint slices, GLint loops)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
inner is the inner radius of the disk (may be 0).
outer is the outer radius of the disk.

GLU sphere:

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices,
GLint stacks)
```

quad identifies the quadrics object you previously created with `gluNewQuadric`
radius is the radius of the sphere.

The GLUT objects

Models provided by GLUT are more oriented to geometric solids, except for the teapot object. They do not have as wide a usage in general situations because they are of fixed shape and many cannot be modeled with varying degrees of complexity. They also do not include shapes that can readily be adapted to general modeling situations. Finally, there is no general way to create a texture map for these objects, so it is more difficult to make scenes using them have stronger visual interest. The GLUT models include a cone (`glutSolidCone`), cube (`glutSolidCube`), dodecahedron (12-sided regular polyhedron, `glutSolidDodecahedron`), icosahedron (20-sided regular polyhedron, `glutSolidIcosahedron`), octahedron (8-sided regular polyhedron, `glutSolidOctahedron`), a sphere (`glutSolidSphere`), a teapot (the Utah teapot, an icon of computer graphics sometimes called the "teapotahedron", `glutSolidTeapot`), a tetrahedron (4-sided regular polyhedron, `glutSolidTetrahedron`), and a torus (`glutSolidTorus`). There are also wireframe versions of each of the GLUT solid objects.

The GLUT primitives include both solid and wireframe versions. Each object has a canonical position and orientation, typically being centered at the origin and lying within a standard volume and, if it has an axis of symmetry, that axis is aligned with the z-axis. As with the GLU standard primitives, the GLUT cone, sphere, and torus allow you to specify the granularity of the primitive's modeling, but the others do not. You should not take the term "solid" for the GLUT objects too seriously, however; they are not actually solid but are simply bounded by polygons. If you clip them you will find that they are, in fact, hollow.

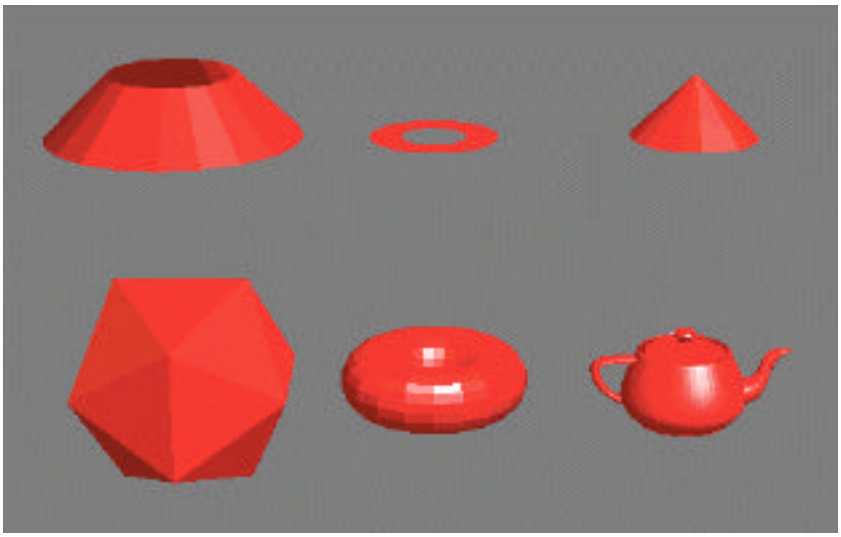


Figure 3.5: several GLU and GLUT objects as described in the text

If you have GLUT with your OpenGL, you should check the GLUT manuals for the details on these solids and on many other important capabilities that GLUT will add to your OpenGL system. If you do not already have it, you can download the GLUT code from the OpenGL Web site for many different systems and install it in your OpenGL area so you may use it readily with your system.

Selections from the overall collection of GLU and GLUT objects are shown in Figure 3.5 to show the range of items you can create with these tools. From top left and moving clockwise, we see a gluCylinder, a gluDisk, a glutSolidCone, a glutSolidIcosahedron, a glutSolidTorus, and a glutSolidTeapot. You should think about how you might use various transformations to create other figures from these basic parts.

An example

Our example for this module is quite simple. It is the heart of the `display()` function for a simple application that displays the built-in sphere, cylinder, dodecahedron, torus, and teapot provided by OpenGL and the GLU and GLUT toolkits. In the full example, there are operations that allow the user to choose the object and to control its display in several ways, but for this example we will only focus on the models themselves, as provided through a switch() statement such as might be used to implement a menu selection. This function is not complete, but would need the addition of viewing and similar functionality that is described in the chapter on viewing and projection.

```
void display( void )
{
   GLUquadric *myQuad;
   GLdouble radius = 1.0;
   GLint slices, stacks;
   GLint nsides, rings;

   ...

   switch (selectedObject) {
      case (1): {
         myQuad=gluNewQuadric();
```

```
            slices = stacks = resolution;
            gluSphere( myQuad , radius , slices , stacks  );
            break;
            }
         case (2): {
            myQuad=gluNewQuadric();
            slices = stacks = resolution;
            gluCylinder( myQuad, 1.0, 1.0, 1.0, slices, stacks );
            break;
            }
         case (3): {
            glutSolidDodecahedron(); break;
            }
         case (4): {
            nsides = rings = resolution;
            glutSolidTorus( 1.0, 2.0, nsides, rings);
            break;
            }
         case (5): {
            glutSolidTeapot(2.0); break;
            }
         }
      ...
}
```

A word to the wise...

One of the differences between student programming and professional programming is that students are often asked to create applications or tools for the sake of learning creation, not for the sake of creating working, useful things. The graphics primitives that are the subject of the first section of this module are the kind of tools that students are often asked to use, because they require more analysis of fundamental geometry and are good learning tools. However, working programmers developing real applications will often find it useful to use pre-constructed templates and tools such as the GLU or GLUT graphics primitives. You are encouraged to use the GLU and GLUT primitives whenever they can save you time and effort in your work, and when you cannot use them, you are encouraged to create your own primitives in a way that will let you re-use them as your own library and will let you share them with others.

Transformations in OpenGL

In OpenGL, there are only two kinds of transformations: projection transformations and modelview transformations. The latter includes both the viewing and modeling transformations. We have already discussed projections and viewing, so here we will focus on the transformations used in modeling.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. In OpenGL, these are applied with the built-in functions (actually function sets) `glRotate`, `glTranslate`, and `glScale`, respectively. As we have found with other OpenGL function sets, there are different versions of each of these, varying only in the kind of parameters they take.

The `glRotate` function is defined as

```
glRotatef(angle, x, y, z)
```

where `angle` specifies the angle of rotation, in degrees, and `x`, `y`, and `z` specify the coordinates of a vector, all as floats (`f`). There is another rotation function `glRotated` that operates in

exactly the same way but the arguments must all be doubles (d). The vector specified in the parameters defines the fixed line for the rotation. This function can be applied to any matrix set in glMatrixMode, allowing you to define a rotated projection if you are in projection mode or to rotate objects in model space if you are in modelview mode. You can use glPushMatrix and glPopMatrix to save and restore the unrotated coordinate system.

This rotation follows the right-hand rule, so the rotation will be counterclockwise as viewed from the direction of the vector (x,y,z). The simplest rotations are those around the three coordinate axes, so that glRotate(angle, 1., 0., 0.) will rotate the model space around the X-axis.

The glTranslate function is defined as

```
glTranslatef(Tx, Ty, Tz)
```

where Tx, Ty, and Tz specify the coordinates of a translation vector as floats (f). Again, there is a translation function glTranslated that operates exactly the same but has doubles (d) as arguments. As with glRotate, this function can be applied to any matrix set in glMatrixMode, so you may define a translated projection if you are in projection mode or translated objects in model space if you are in modelview mode. You can again use glPushMatrix and glPopMatrix to save and restore the untranslated coordinate system.

The glScale function is defined as

```
glScalef(Sx, Sy, Sz)
```

where Sx, Sy, and Sz specify the coordinates of a scaling vector as floats (f). Again, there is a translation function glScaled that operates exactly the same but has doubles (d) as arguments. As above, this function can be applied to any matrix set in glMatrixMode, so you may define a scaled projection if you are in projection mode or scaled objects in model space if you are in modelview mode. You can again use glPushMatrix and glPopMatrix to save and restore the unscaled coordinate system. Because scaling changes geometry in non-uniform ways, a scaling transformation may change the normals of an object. If scale factors other than 1.0 are applied in modelview mode and lighting is enabled, automatic normalization of normals should probably also be enabled. See the chapter on lighting for details.

OpenGL provides a number of facilities for manipulating transformations. As we will see in more detail in the chapter on mathematical fundamentals, a transformation for 3D computer graphics is represented by a 4x4 array, which in turn is stored as an array of 16 real numbers. You may save the current modelview matrix with the function glGetFloatv(GL_MODELVIEW_MATRIX, params) where params is an array GLfloat params[16]. You do not restore the modelview matrix directly, but if your transformation mode is set to modelview with the function glMatrixMode(GL_MODELVIEW), you can multiply the current modelview matix by myMatrix, saved as a 16-element array, with the function glMultMatrix(myMatrix). You can do similar manipulations of the OpenGL projection matrix. This kind of operation requires you to be comfortable with expressing and manipulating transformations as matrices, but OpenGL provides enough transformation tools that it's rare to need to handle transformations this way.

As we saw earlier in the chapter, there are many transformations that go into defining exactly how a piece of geometry is presented in a graphics scene. When we consider the overall order of transformations for the entire model, we must consider not only the modeling transformations but also the projection and viewing transformations. If we consider the total sequence of transformations in the order in which they are specified, we will have the sequence:

```
P   V   T0  T1  …   Tn  Tn+1    …   Tlast
```

with P being the projection transformation, V the viewing transformation, and T0, T1, … Tlast the transformations specified in the program to model the scene, in order (T1 is first, Tlast is last and is closest to the actual geometry). The projection transformation is defined in the reshape function; the viewing transformation is defined in the init function, in the reshape function, or at the beginning of the display function so it is defined at the beginning of the modeling process. But the sequence in which the transformations are applied is actually the reverse of the sequence above: Tlast is actually applied first, and V and finally P are applied last. You need to understand this sequence very well, because it's critical to understand how you build complex, hierarchical models.

Code examples for transformations

Simple transformations:
All the code examples use a standard set of axes, which are not included here, and the following definition of the simple square:

```
void square (void)
{
   typedef GLfloat point [3];
   point v[8] = {{12.0, -1.0, -1.0},
                 {12.0, -1.0,  1.0},
                 {12.0,  1.0,  1.0},
                 {12.0,  1.0, -1.0} };

   glBegin (GL_QUADS);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
   glEnd();
}
```

To display the simple rotations example, we use the following display function:

```
void display( void )
{  int i;
   float theta = 0.0;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<8; i++) {
      glPushMatrix();
      glRotatef(theta, 0.0, 0.0, 1.0);
      if (i==0)
         glColor3f(1.0, 0.0, 0.0);
      else
         glColor3f(1.0, 1.0, 1.0);
      square();
      theta += 45.0;
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple translations example, we use the following display function:

```
void display( void )
{  int i;
```

```
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<=12; i++) {
      glPushMatrix();
      glTranslatef(-2.0*(float)i, 0.0, 0.0);
      if (i==0) glColor3f(1.0, 0.0, 0.0);
         else  glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple scaling example, we use the following display function:

```
void display( void )
{  int i;
   float s;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<6; i++) {
      glPushMatrix();
      s = (6.0-(float)i)/6.0;
      glScalef( s, s, s );
      if (i==0)
         glColor3f(1.0, 0.0, 0.0);
      else
         glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

Transformation stacks: The OpenGL functions that are used to manage the transformation stack are `glPushMatrix()` and `glPopMatrix()`. Technically, they apply to the stack of whatever transformation is the current matrix mode, and the `glMatrixMode` function with parameters `GL_PROJECTION` and `GL_MODELVIEW` sets that mode. We only rarely want to use a stack of projection transformations (and in fact the stack of projections can only hold two transformations) so we will almost always work with the stack of modeling/viewing transformation. The rabbit head example was created with the display function given below. This function makes the stack operations more visible by using indentations; this is intended for emphasis in the example only and is not standard programming practice in graphics. Note that we have defined only very simple display properties (just a simple color) for each of the parts; we could in fact have defined a much more complex set of properties and have made the parts much more visually interesting. We could also have used a much more complex object than a simple `gluSphere` to make the parts much more structurally interesting. The sky's the limit…

```
void display( void )
{
// Indentation level shows the level of the transformation stack
// The basis for this example is the unit gluSphere; everything
// else is done by explicit transformations

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      // model the head
```

```
        glColor3f(0.4, 0.4, 0.4);    //   dark gray head
        glScalef(3.0, 1.0, 1.0);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
     glPopMatrix();
     glPushMatrix();
        // model the left eye
        glColor3f(0.0, 0.0, 0.0);    //   black eyes
        glTranslatef(1.0, -0.7, 0.7);
        glScalef(0.2, 0.2, 0.2);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
     glPopMatrix();
     glPushMatrix();
        // model the right eye
        glTranslatef(1.0, 0.7, 0.7);
        glScalef(0.2, 0.2, 0.2);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
     glPopMatrix();
     glPushMatrix();
        // model the left ear
        glColor3f(1.0, 0.6, 0.6);    //   pink ears
        glTranslatef(-1.0, -1.0, 1.0);
        glRotatef(-45.0, 1.0, 0.0, 0.0);
        glScalef(0.5, 2.0, 0.5);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
     glPopMatrix();
     glPushMatrix();
        // model the right ear
        glColor3f(1.0, 0.6, 0.6);    //   pink ears
        glTranslatef(-1.0, 1.0, 1.0);
        glRotatef(45.0, 1.0, 0.0, 0.0);
        glScalef(0.5, 2.0, 0.5);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
     glPopMatrix();
     glutSwapBuffers();
  }
```

In OpenGL, the stack for the modelview matrix is to be at least 32 deep, but this can be inadequate to handle some complex models if the hierarchy is more than 32 layers deep. In this case, as we mentioned in the previous chapter, you need to know that a transformation is a 4x4 matrix of `GLfloat` values that is stored in a single array of 16 elements. You can create your own stack of these arrays that can have any depth you want, and then push and pop transformations as you wish on that stack. To deal with the modelview transformation itself, there are functions that allow you to save and to set the modelview transformation as you wish. You can capture the current value of the transformation with the function

```
glGetFloatv(GL_MODELVIEW_MATRIX, viewProj);
```

(here we have declared `GLfloat viewProj[16]`), and you can use the functions

```
glLoadIdentity();
glMultMatrixf( viewProj );
```

to set the current modelview matrix to the value of the matrix `viewProj`, assuming that you were in modelview mode when you execute these functions.

Inverting the eyepoint transformation

In an example somewhat like the more complex eye-following-helicopter example above, we built a small program in which the eye follows a red sphere at a distance of 4 units as the sphere flies in a circle above some geometry. In this case, the geometry is a cyan plane on which are placed several cylinders at the same distance from the center as the sphere flies, along with some coordinate axes. A snapshot from this very simple model is shown in Figure 3.6. The display function code that implements this viewing is shown after the figure; you will note that the display function begins with the default view and is followed by the transformations

translate by –4 in Z
translate by –5 in X and –.75 in Y
rotate by -theta around Y

that are the inverse of the cylinder placement and eye placement, first for the sphere

rotate by theta around Y
translate by 5 in X and .75 in Y

and then for the eye to lie 4 units in Z from the sphere. Note that this means, for example, that when the sphere is on the X-axis, for example, the eye is 4 units from the sphere in the Z-direction. There is no explicit transformation in the code, however; the inverse is from the position of the eyepoint in the scene graph, because the eye point is never set relative to the sphere in the code.

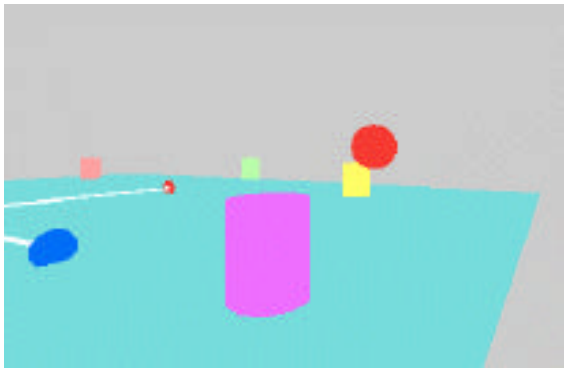


Figure 3.6: the eye following a sphere flying over some cylinders on a plane

```
void display( void )
{
   ...
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   // Define eye position relative to the sphere it is to follow
   // place eye in scene with default definition
   gluLookAt(0.,0.,0., 0.,0.,-1., 0.,1.,0.);

   // invert transformations that place the eye
   glTranslatef(0.,0.,-4.);
   glTranslatef(-5.,-0.75,0.);
   glRotatef(-theta,0.,1.,0.);

   // draw the sphere we're following around...
   glPushMatrix();
   glRotatef(theta, 0., 1., 0.);
   glTranslatef(5., 0.75, 0.);
```

```
   glColor3f(1., 0., 0.);
   myQuad = gluNewQuadric();
   gluSphere(myQuad, .25, 20, 20);
   glPopMatrix();

   // draw whatever geometry the sphere is flying over...
   ...

     glutSwapBuffers();
}
```

Creating display lists

In OpenGL, graphics objects can be compiled into what is called a *display list*, which will contain the final geometry of the object as it is ready for display. OpenGL display lists are named by nonzero unsigned integer values (technically, `GLuint` values) and there are several tools available in OpenGL to manage these name values. We will assume in a first graphics course that you will not need many display lists and that you can manage a small number of list names yourself, but if you begin to use a number of display lists in a project, you should look into the `glGenLists`, `glIsList`, and `glDeleteLists` functions to help you manage the lists properly. Sample code and a more complete explanation is given below.

Display lists are relatively easy to create in OpenGL. First, choose an unsigned integer (often you will just use small integer constants, such as 1, 2, ...) to serve as the name of your list. Then before you create the geometry for your list, call the function `glNewList`. Code whatever geometry you want into the list, and at the end, call the function `glEndList`. Everything between the new list and the end list functions will be executed whenever you call `glCallList` with a valid list name as parameter. All the operations between `glNewList` and `glEndList` will be carried out, and only the actual set of instructions to the drawing portion of the OpenGL system will be saved. When the display list is executed, then, those instructions are simply sent to the drawing system; any operations needed to generate these instructions are omitted.

Because display lists are often defined only once, it is common to create them in the `init()` function or in a function called from within `init()`. Some sample code is given below, with most of the content taken out and only the display list operations left.

```
   void Build_lists(void) {
      glNewList(1, GL_COMPILE);
         glBegin(GL_TRIANGLE_STRIP);
            glNormal3fv(...); glVertex3fv(...);
            ...
         glEnd();
      glEndList();
   }

   static void Init(void) {
      ...
      Build_lists();
      ...
   }

   void Display(void) {
      ...
      glCallList(1);
      ...
```

```
}
```

You will note that the display list was created in `GL_COMPILE` mode, and it was not executed (the object was not displayed) until the list was called. It is also possible to have the list displayed as it is created if you create the list in `GL_COMPILE_AND_EXECUTE` mode.

Chapter 4: Mathematics for Modeling

The primary mathematical background needed for computer graphics programming is 3D analytic geometry. It is unusual to see a course with this title, however, so most students pick up bits and pieces of mathematics background that fill this in. One of the common sources of the background is introductory physics; another is multivariate calculus. Neither of these is a common requirement for computer graphics, however, so here we will outline the general concepts we will use in these notes.

Coordinate systems

The set of real numbers—often thought of as the set of all possible distances between points—is a mathematical abstraction that is effectively modeled as a Euclidean straight line with two uniquely-identified points. One point is identified with the number 0.0 (we will write all real numbers with decimals, to meet the expectations of programming languages), called the origin, and the other is identified with the number 1.0, which we call the unit point. The direction of the line from 0.0 to 1.0 is called the positive direction; the opposite direction of the line is called the negative direction. These directions identify the parts of the lines associated with positive and negative numbers, respectively.

If we have two straight lines that are perpendicular to each other and meet in a point, we can define that point to be the origin for both lines, and choose two points the same distance from the origin on each line as the unit points. A distance unit is defined to be used by each of the two lines, and the points at this distance from the intersection point are marked, one to the right of the intersection and one above it. This gives us the classical 2D coordinate system, often called the Cartesian coordinate system. The vectors from the intersection point to the right-hand point (respectively the point above the intersection) are called the X- and Y-direction vectors and are indicated by **i** and **j** respectively. Points in this system are represented by an ordered pair of real numbers, (X, Y), and this is probably the most familiar coordinate system to most people. These points may also be represented by a vector <X,Y> from the origin to the point, and this vector may be expressed in terms of the direction vectors as X**i**+Y**j**.

In 2D Cartesian coordinates, any two lines that are not parallel will meet in a point. The lines make four angles when they meet, and the acute angle is called the angle between the lines. If two line segments begin at the same point, they make a single angle that is called the angle between the line segments. These angles are measured with the usual trigonometric functions, and we assume that the reader will have a modest familiarity with trigonometry. Some of the reasons for this assumption can be found in the discussions below on polar and spherical coordinates, and in the description of the dot product and cross product. We will discuss more about the trigonometric aspects of graphics when we get to that point in modeling or lighting.

The 3D world in which we will do most of our computer graphics work is based on 3D Cartesian coordinates that extend the ideas of 2D coordinates above. This is usually presented in terms of three lines that meet at a single point, which is identified as the origin for all three lines and is called the *origin*, that have their unit points the same distance from that point, and that are mutually perpendicular. Each point represented by an ordered triple of real numbers (x,y,z). The three lines correspond to three unit direction vectors, each from the origin to the unit point of its respective line; these are named **i**, **j**, and **k** for the X-, Y-, and Z-axis, respectively, and are called the canonical basis for the space, and the point can be represented as x**i**+y**j**+z**k**. Any ordered triple of real numbers is identified with the point in the space that lies an appropriate distance from the two-axis planes, with the first (x) coordinate being the distance from the Y-Z plane, the second (y) coordinate being the distance from the X-Z plane, and the third (z) coordinate being the distance from the X-Y plane. This is all illustrated in Figure 4.1.

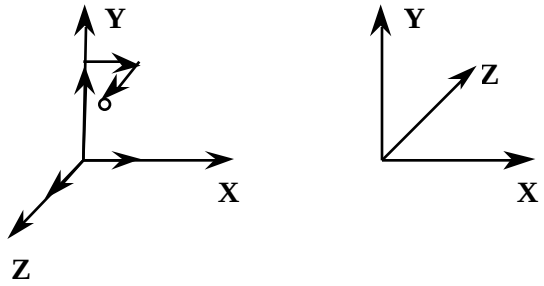


Figure 4.1: right-hand coordinate system with origin (left) and with a point identified by its coordinates; left-hand coordinate system (right)

3D coordinate systems can be either right-handed or left-handed: the third axis can be the cross product of the first two axes, or it can be the negative of that cross product, respectively. (We will talk about cross products a little later in this chapter.) The "handed-ness" comes from a simple technique: if you hold your hand in space with your fingers along the first axis and curl your fingers towards the second axis, your thumb will point in a direction perpendicular to the first two axis. If you do this with the right hand, the thumb points in the direction of the third axis in a right-handed system. If you do it with the left hand, the thumb points in the direction of the third axis in a left-handed system.

Some computer graphics systems use right-handed coordinates, and this is probably the most natural coordinate system for most uses. For example, this is the coordinate system that naturally fits electromagnetic theory, because the relationship between a moving current in a wire and the magnetic field it generates is a right-hand coordinate relationship. The modeling in Open GL is based on a right-hand coordinate system.

On the other hand, there are other places where a left-handed coordinate system is natural. If you consider a space with a standard X-Y plane as the front of the space and define Z as the distance back from that plane, then the values of Z naturally increase as you move back into the space. This is a left-hand relationship.

Points, lines, and line segments

In this model, any real number is identified with the unique point on the line that is

- at the distance from the origin which is that number times the distance from 0.0 to 1.0, and
- in the direction of the number's sign.

We have heard that a line is determined by two points; let's see how that can work. Let the first point be $P0 = (X0, Y0, Z0)$ and the second point be $P1 = (X1, Y1, Z1)$. Let's call $P0$ the origin and $P1$ the unit point. Points on the segment are obtained by starting at the "first" point $P0$ offset by a fraction of the difference vector $P1$-$P0$. Then any point $P = (X, Y, Z)$ on the line can be expressed in vector terms by

$$P = P0 + t(P1 - P0) = (1 - t)P0 + tP1$$

for a single value of a real variable `t`. This computation is actually done for each coordinate, with a separate equation for each of X, Y, and Z as follows:

$$X = X0 + t(X1 - X0) = (1 - t)X0 + tX1$$

$$Y = Y0 + t(Y1 - Y0) = (1 - t)Y0 + tY1$$

$$Z = Z0 + t(Z1 - Z0) = (1 - t)Z0 + tZ1$$

Thus any line segment can be determined by a single parameter, which is why a line is called a one–dimensional object. Points along the line are determined by values of the parameter, as

illustrated in Figure 4.2 below that shows the coordinates of the points along a line segment determined by value of `t` from 0 to 1 in increments of 0.25.

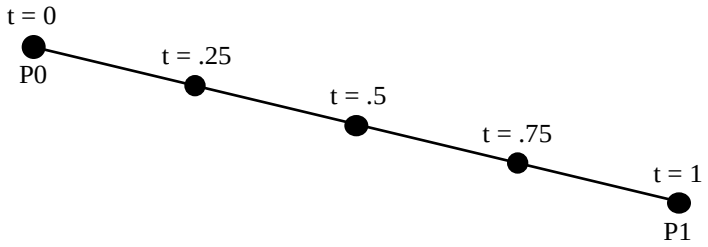


Figure 4.2: a parametric line segment with points determined by some values of the parameter

This representation for a line segment (or an entire line, if you place no restrictions on the value of *t*) also allows you to compute intersections involving lines. The reverse concept is also useful, so if you have a known point on the line, you can calculate the value of the parameter *t* that would produce that point. For example, if a line intersects a plane or another geometric object at a point *Q*, a vector calculation of the form $P0 + t\ (P1 - P0) = Q$ would allow you to calculate the value of the parameter *t* that gives the intersection point on the line. This calculation might involve only a single equation or all three equations, depending on the situation, but your goal is to compute the value of *t* that represents the point in question. This is often the basis for geometric computations such as the intersection of a line and a plane.

As an example of this, we'll take two points and calculate the parametric equations for the line. Let P0 = (3.0, 4.0, 5.0) and P1 = (5.0, -1.5, 4.0). Then P1 - P0 = (2.0, -5.5, -1.0), so the equations of the line are

x = 3.0 + 2.0t
y = 4.0 – 5.5t
z = 5.0 – t

If we consider a plane 6.0x – 2.0y + 1.5z – 4.0 = 0.0, the point where the line intersects the plane is given by 6.0(3.0 + 2.0t) – 2.0(4.0 – 5.5t) + 1.5(5.0 – t) – 4.0 = 0.0. Combining terms yields 21.5t – 13.5 = 0, or t = 13.5/21.5 = 27/43, from which the intersection point is (183/43, –41/86, 188/43). Of course, you will rarely do this kind of calculation manually, but will code up such computations as needed in your program.

Distance from a point to a line

As an application of the concept of parametric lines, let's see how we can compute the distance from a point in 3-space to a line. If the point is P0=(u,v,w) and the line is given by parametric equations

$$x = a + bt$$

$$y = c + dt$$

$$z = e + ft$$

then for any point P=(x,y,z) on the line, the square of thedistance from P to P0 is given by

$$(a + bt - u)^2 + (c + dt - v)^2 + (e + ft - w)^2$$

which is a quadratic equation in *t*. This quadratic is minimized by taking its derivative and looking for the point where the derivative is 0:

$$2b(a + bt - u) + 2d(c + dt - v) + 2f(e + ft - w) = 0$$

which is a simple linear equation in t, and its unique solution for t allows you to calculate the point P on the line which is nearest point P0.

Line segments and parametric curves

In standard Euclidean geometry, two points determine a line as we noted above. In fact, in the same way we talked about any line having unique origin identified with 0.0 and unit point identified with 1.0, a line segment—the points on the line between these two particular points—can be identified as the points corresponding to values between 0 and 1. It is done by much the same process as we used to illustrate the 1-dimensional nature of a line above. That is, just as in the discussion of lines above, if the two points are $P0$ and $P1$, we can identify any point between them as $P = (1-t)P0 + tP1$ for a unique value of t between 0 and 1. This is called the parametric form for a line segment

The line segment gives us an example of determining a continuous set of points by functions from the interval [0.1] to 3-space. In general, if we consider any set of continuous functions $x(t)$, $y(t)$, and $z(t)$ that are defined on [0,1], the set of points they generate is called a *parametric curve* in 3-space. There are some very useful applications of such curves. For example, you can display the locations of a moving point in space, you can compute the positions along a curve from which you will view a scene in a fly-through, or you can describe the behavior of a function of two variables on a domain that lies on a curve in 2-space.

Vectors

Vectors in 3-space are triples of real numbers written as $< a,\ b,\ c >$. These may be identified with points, or they may be viewed as representing the motion needed to go from one point to another in space. The latter viewpoint will be one we use often.

The *length* of a vector is defined as the square root of the sum of the squares of the vector's components, written $\| < a,b,c > \| = \sqrt{a^2 + b^2 + c^2}$. A *unit* vector is a vector whose length is 1, and unit vectors are very important in a number of modeling and rendering computations; basically a unit vector can be treated as a pure direction. If $V =< a,b,c >$ is any vector, we can make it a unit vector by dividing each of its components by its length: $\left\langle a/\|V\|, b/\|V\|, c/\|V\| \right\rangle$. Doing this is called *normalizing* the vector.

Dot and cross products of vectors

There are two computations on vectors that we will need to understand, and sometimes to perform, in developing the geometry for our graphic images. The first is the *dot product* of two vectors. This produces a single real value that represents the projection of one vector on the other and its value is the product of the lengths of the two vectors times the cosine of the angle between them. The dot product computation is quite simple: it is simply the sum of the componentwise products of the vectors. If the two vectors A and B are

$$A =< X1,Y1,Z1 >$$
$$B =< X2,Y2,Z2 >$$

then their dot product is computed as

$$A \cdot B = X1 * X2 + Y1 * Y2 + Z1 * Z2 \ .$$

Note that a simple consequence of the definition is that the length of any vector A is the square root of the dot product $A \bullet A$ of the vector with itself.

The observation about the cosine of the angle between the vectors is very important. If two vectors are parallel, the dot product is simply the product of their lengths, but if they are orthogonal—the angle between them is 90°—then the dot product is zero. If the angle between them is acute, then the dot product will be positive, no matter what the orientation of the vectors, because the cosine of any angle between –90° and 90° is positive; if the angle between them is obtuse, then the dot product will be negative. Note that another useful application of the fact is that you can compute the angle between two vectors if you know the vectors' lengths and dot products. These observations about the dot product is very useful in a number of graphical computations.

The relationship between the dot product and the cosine of the included angle is very important. We can take advantage of it to look at the component of any vector that lies in the direction of another vector, which we call the *projection* of one vector on another. As we see in Figure 4.3, with any two vectors we can construct a right triangle in which one side is one of the vectors and the other is the projection of the first on the second. Because $U \cdot V = \|U\| \ \|V\| \cos(\Theta)$, and because the projection of U onto V is given by $\|U\| \cos(\Theta)$, we see that the projection of U onto V is actually $U \cdot V / \|V\|$. This is especially useful when V is a unit vector because then the dot product alone is the projection, and this is one of the reasons for normalizing the vectors we use.

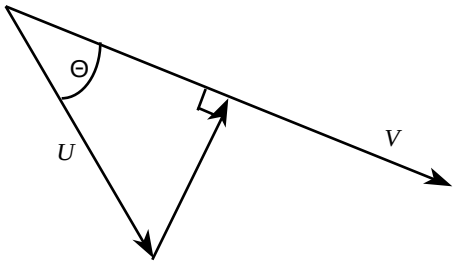


Figure 4.3: diagram of the projection of U onto V

The second computation is the *cross product*, or vector product, of two vectors. The cross product of two vectors is a third vector that is perpendicular to each of the original vectors and whose length is the product of the two vector lengths times the sin of the angle between them. Thus if two vectors are parallel, the cross product is zero; if they are orthogonal, the cross product has length equal to the product of the two lengths; if they are both unit vectors, the cross product is the sine of the included angle. The computation of the cross product can be expressed as the determinant of a matrix whose first row is the three standard unit vectors, whose second row is the first vector of the product, and whose third row is the second vector of the product. Denoting the unit direction vectors in the X, Y, and Z directions as i, j, and k, as above, we can express the cross product of two vectors $< a,b,c >$ and $< u,v,w >$ in terms of a determinant:

$$< a,b,c > \times < u,v,w > = \det\begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix} = i \det\begin{vmatrix} b & c \\ v & w \end{vmatrix} - j \det\begin{vmatrix} a & c \\ u & w \end{vmatrix} + k \det\begin{vmatrix} a & b \\ u & v \end{vmatrix} = < bw - cv, cu - aw, av - bu >$$

As an example here, consider the two points we saw earlier, but treat them as vectors: u = <3.0, 4.0, 5.0> and v = <5.0, -1,5, 4.0>. Then the length of u is the square root of $u \bullet u$, or 7.071, and the length of v is the square root of $v \bullet v$, or 6.576. Then we see that $u \bullet v$ = 15.0 – 6.0 + 20.0 = 29.0, and the cosine of the angle between u and v is 29.0/(7.071*6.576) or 0.624. Further, the cross product of the two vectors is computed as

$$u \times v = \det\begin{vmatrix} i & j & k \\ 3 & 4 & 5 \\ 5 & -1.5 & 4 \end{vmatrix} = i\begin{vmatrix} 4 & 5 \\ -1.5 & 4 \end{vmatrix} - j\begin{vmatrix} 3 & 5 \\ 5 & 4 \end{vmatrix} + k\begin{vmatrix} 3 & 4 \\ 5 & -1.5 \end{vmatrix}$$

Carrying out the 2x2 determinants gives us u x v = 23.5 i + 13.0 j – 24.5 k as the cross product. You should check to see that this product is orthogonal to both u and v by computing the dot products, which should be 0.

The cross product has a "handedness" property and is said to be a right-handed operation. That is, if you align the fingers of your right hand with the direction of the first vector and curl your fingers towards the second vector, your right thumb will point in the direction of the cross product. Thus the order of the vectors is important; if you reverse the order, you reverse the sign of the product (recall that interchanging two rows of a determinant will change its sign), so the cross product operation is not commutative. As a simple example, with **i**, **j**, and **k** as above, we see that **i** x **j** = **k** but that **j** x **i** = -**k**. In general, if you consider the arrangement of Figure 4.4, if you think of the three direction vectors as being wrapped around as if they were visible from the first octant of 3-space, the product of any two is the third direction vector if the letters are in counterclockwise order, and the negative of the third if the order is clockwise. Note also that the cross product of two collinear vectors (one of the vectors is a constant multiple of the other) will always be zero, so the geometric interpretation of the cross product does not apply in this case.

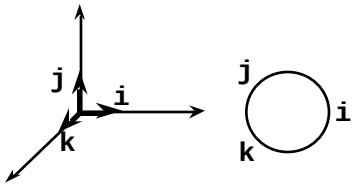


Figure 4.4: the direction vectors in order

The cross product can be very useful when you need to define a vector perpendicular to two given vectors; the most common application of this is defining a normal vector to a polygon by computing the cross product of two edge vectors. For a triangle as shown in Figure 4.5 with

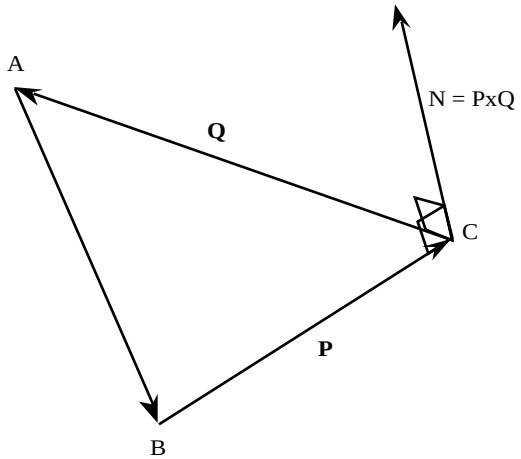


Figure 4.5: the normal to a triangle as the cross product of two edges

vertices A, B, and C in order counterclockwise from the "front" side of the triangle, the normal vector can be computed by creating the two difference vectors $P = C - B$ and $Q = A - C$, and computing the cross product as P x Q to yield a vector N normal to the plane of the triangle. In fact, we can say more than this; the cross product of two vectors is not only perpendicular to the plane defined by those vectors, but its length is the product of their lengths times the sine of the angle between them. As we shall see in the next section, this normal vector, and any point on the triangle, allow us to generate the equation of the plane that contains the triangle. When we need to use this normal for lighting we will need to normalize it, or make it a unit-length vector as we described above, but that is easily done by calculating the vector's length and dividing each component by that length.

Reflection vectors

There are several times in computer graphics where it is important to calculate a vector that is a reflection of another vector in some surface. One example is in specular light calculations; we will see later that the brightness of specular light at a point (shiny light reflected from the surface similarly to the way a mirror reflects light) will depend on the angle between the vector to the eye from that point and the reflection of the vector from that point to the light. Another example is in any model where objects hit a surface and are reflected from it, where the object's velocity vector after the bounce is the reflection of its incoming velocity vector. In these cases, we need to know the normal to the surface at the point where the vector to be reflected hits the surface, and the calculation is fairly straightforward. Figure 4.6 shows the situation we are working with in the case of reflected velocities; the situation with light is similar except that the P vector is directed outwards instead of inwards.

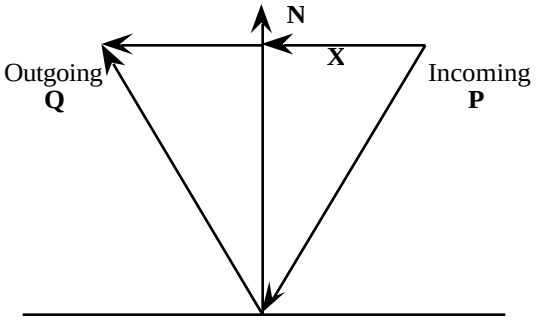


Figure 4.6: Incoming vector, outgoing vector, and normal vector

In this figure, let N^* be the vector that Q makes when it is projected on N; then $N^* = -(N \bullet P)N$. We see that $X = P + N^* = P - (N \bullet P)N$. But $Q + P = 2X$, so $Q = 2(P - (N \bullet P)N) - P$, from which $Q = P - 2(N \bullet P)N$. This is actually an easy calculation and the code is left to the reader.

Transformations

In the previous two chapters we discussed transformations rather abstractly: as functions that operate on 3D space to produce given effects. In the spirit of this chapter, however, we describe how these functions are represented concretely for computation and, in particular, the representation of each of the basic scaling, rotation, and transformation matrices.

To begin, we recall that we earlier introduced the notion of homogeneous coordinates for points in 3D space: we identify the 3D point (x, y, z) with the homogeneous 4D point (x, y, z, 1). The transformations in 3D computer graphics are all linear functions on 4D space and so may be represented as 4x4 matrices:

$$T = \begin{matrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{matrix}$$

Applying two transformations in order, or composing the transformations, is accomplished by multiplying the transformations' matrices. So if we have transformations *S* and *T*, represented by arrays {*S*[*i*][*j*]} and {*T*[*i*][*j*]}, respectively, then the composition of the transformations is given by *C* = *S***T*; in terms of code for multiplying the matrices this is

```
for (int i = 0; i < 4; i++ )
   for (int j = 0; j < 4; j++ ) {
      C[i][j] = 0.;
      for (int k = 0; k < 4; k++)
         C[i][j] += S[i][k]*T[k][j];
   }
```

This is straightforward but fairly slow; you may be able to find ways to speed it up. Geometrically, this treats the matrices as sets of vectors: the left-hand matrix is composed of row vectors and the right-hand matrix of column vectors. The the product of the matrices is composed of the dot products of each row matrix from the left by each column matrix on the right.

The effect of a transformation on a vector is given by multiplying the transformation, as a matrix, on the left of the point, stored as a column vector. This uses the same code as above except that because we are multipying a 4x4 matrix on the left by a 4x1 matrix on the right, the index *j* has only the value 0 rather than the values 0 through 3.

So with this background, let's proceed to consider how the basic transformations look as matrices. For scaling, the OpenGL function `glScalef(sx, sy, sz)` is expressed as

$$\begin{matrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

For translation, the OpenGL function `glTranslatef(tx, ty, tz)` is expressed as

$$\begin{matrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{matrix}$$

For rotation, we have a more complex situation. OpenGL allows you to define a rotation around any given line with the function `glRotatef(angle, x, y, z)` where *angle* is the amount or rotation (in degrees), and <*x*, *y*, *z*> is the direction vector of the line around which the rotation is to be done. We can write a matrix for the general rotation, but before we do that, let's look at the simpler rotations around the coordinate axes. For the rotation around the *X*-axis, `glRotatef(angle, 1., 0., 0.)`, the matrix is as follows; note that the first component, the *X*-component, is not changed by this matrix.

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos(angle) & -\sin(angle) & 0 \\ 0 & \sin(angle) & \cos(angle) & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

For the rotation around the *Z*-axis, `glRotatef(angle, 0., 0., 1.)`, the matrix is much the same as the *X*-axis rotation but with a different fixed space.

cos(*angle*)	-sin(*angle*)	0	0
sin(*angle*)	cos(*angle*)	0	0
0	0	1	0
0	0	0	1

For the Y-axis, there is a difference because the cross produce of the X- and Z-axes is in the opposite direction to the Y-axis. This means that the angle relative to the Y-axis is the negative of the angle relative to the cross product, giving us a change in the sign of the sine function. So the matrix for glRotatef(angle, 0., 1., 0.) is:

cos(*angle*)	0	sin(*angle*)	0
0	1	0	0
-sin(*angle*)	0	cos(*angle*)	0
0	0	0	1

The formula for a rotation around an arbitrary line is more complex and is given in the OpenGL manual, so we will not present it here.

Planes and half-spaces

We saw above that a line could be defined in terms of a single parameter, so it is often called a one-dimensional space. A plane, on the other hand, is a two-dimensional space, determined by two parameters. If we have any two non-parallel lines that meet in a single point, we recall that they determine a plane that can be thought of as all points that are translations of the given point by vectors that are linear combinations of the direction vectors of the two lines. Thus any plane in space is seen as two-dimensional where each of the two lines contributes one of the dimensional components. In practice, we usually don't have two lines in the plane but have three points in the plane that do not lie in a single straight line, and we get the two lines by letting each of two different pairs of points determine a line. Because each pair of points lies in the plane, so does each of the two lines they generate, and so we have two lines.

In more general terms, let's consider the vector $N = < A,B,C >$ defined as the cross product of the two vectors determined by the two lines. Then N is perpendicular to each of the two vectors and hence to any line in the plane. In fact, this can be taken as defining the plane: the plane is defined by all lines through the fixed point perpendicular to N. If we take a fixed point in the plane, (U,V,W), and a variable point in the plane, (x,y,z), we can use the dot product to express the perpendicular relationship as

$$< A,B,C > \cdot < u-U, y-V, z-W >= 0 .$$

When we expand this dot product we see

$$A(u-X) + B(y-V) + C(z-W) = Ax + By + Cz + (-AU - BV - CW) = 0.$$

This allows us to give an equation for the plane:

$$Ax + By + Cz + D = 0$$

for an appropriate value of D. Thus the coefficients of the variables in the plane equation exactly match the components of the vector normal to the plane—a very useful fact from time to time.

We can readily see that a plane divides 3D space into two parts, but we need to know how to tell which points are in which part. To see this, let's look first at 2D space. Any line divides a plane into two parts, and if we know the equation of the line in the traditional form

$$ax + by + c = 0,$$

then we can determine whether a point lies on, above, or below the line by evaluating the function $f(x,y) = ax + by + c$ and determining whether the result is zero, positive, or negative, respectively. In a similar way, the equation for the plane as defined above does more than just identify the plane; it allows us to determine on which side of the plane any point lies. If we create a function of three variables from the plane equation

$$f(x,y,z) = Ax + By + Cz + D,$$

then the plane consists of all points where $f(x,y,z)=0$. All points (x,y,z) with $f(x,y,z)>0$ lie on one side of the plane, called the positive half-space for the plane, while all points with $f(x,y,z)<0$ lie on the other, called the negative half-space for the plane. We will find that OpenGL uses the four coordinates A, B, C, D to identify a plane and uses the half-space concept to choose displayable points when the plane is used for clipping.

Let's consider an example here that will illustrate both the previous section and this section. If we consider three points A = (1.0, 2.0, 3.0), B = (2.0, 1.0, -1.0), and C = (-1.0, 2.0, 1.0), we can easily see that they do not lie on a single straight line in 3D space. Thus these three points define a plane; let's calculate the plane's equation.

To begin, the difference vectors are $A–B$ = <-1.0, 1.0, 4.0> and $B–C$ = <3.0, -1.0, -2.0> for the original points, so these two vectors applied to any one of the points will determine two lines in the plane. We then compute the cross product $(B–C)$x$(A–B)$ of these two vectors as we outlined above, and get

$$\det\begin{vmatrix} i & j & k \\ 3 & -1 & -2 \\ -1 & 1 & 4 \end{vmatrix} = \langle -2.0, -10.0, 2.0 \rangle$$

Thus the equation of the plane is $-2X - 10Y + 2Z + D = 0$, and putting in the coordinates of B we can calculate the constant D as –12.0, giving a final equation as $-2X - 10Y + 2Z - 12 = 0$. Here any point for which the plane equation yields a positive value lies on the side of the plane in the direction the normal is facing, and any point that yields a negative value lies on the other side of the plane.

Distance from a point to a plane

Just as we earlier defined a way to compute the distance from a point to a line, we also want to be able to compute the distance from a point to a plane. This will be useful when we discuss collision detection later, and may also have other applications.

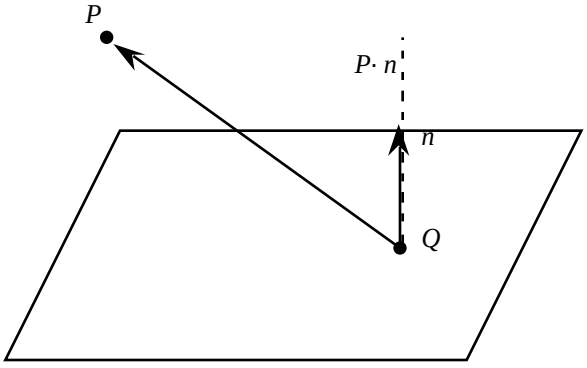


Figure 4.7: the computation of the distance from a point to a plane

Let's consider a plane $Ax+By+Cz+D=0$ with normal vector $N=\langle A,B,C\rangle$, unit normal vector $n=\langle a,b,c\rangle$, an arbitrary point $P=(u,v,w)$. Then select any point $Q=(d,e,f)$ in the plane, and

consider the relationships shown in Figure 4.7. The diagram shows that the distance from the point to the plane is the projection of the vector $P–Q$ on the unit normal vector n. This gives us an easy way to compute this distance, especially since we can choose the point Q any way we wish.

Polygons and convexity

Most graphics systems, including OpenGL, are based on modeling and rendering based on polygons and polyhedra. A *polygon* is a plane region bounded by a sequence of directed line segments with the property that the end of one segment is the same as the start of the next segment, and the end of the last line segment is the start of the first segment. A *polyhedron* is a region of 3–space that is bounded by a set of polygons. Because polyhedra are composed of polygons, we will focus on modeling with polygons, and this will be a large part of the basis for the modeling chapter below.

The reason for modeling based on polygons is that many of the fundamental algorithms of graphics have been designed for polygon operations. In particular, many of these algorithms operate by interpolating values across the polygon; you will see this below in depth buffering, shading, and other areas. In order to interpolate across a polygon, the polygon must be *convex*. Informally, a polygon is complex if it has no indentations; formally, a polygon is complex if for any two points in the polygon (either the interior or the boundary), the line segment between them lies entirely within the polygon.

Because a polygon bounds a region of the plane, we can talk about the interior or exterior of the polygon. In a convex polygon, this is straightforward because the figure is defined by its bounding planes or lines, and we can simply determine which side of each is "inside" the figure. If your graphics API only allows you to define convex polygons, this is all you need consider. In general, though, polygons can be non-convex and we would like to define the concept of "inside" for them. Because this is less simple, we look to convex figures for a starting point and notice that if a point is inside the figure, any ray from an interior point (line extending in only one direction from the point) must exit the figure in precisely one point, while if a point is outside the figure, if the ray hits the polygon it must both enter and exit, and so crosses the boundary of the figure in either 0 or 2 points. We extend this idea to general polygons by saying that a point is inside the polygon if a ray from the point crosses the boundary of the polygon an odd number of times, and is outside the polygon if a ray from the point crosses the boundary of the polygon an even number of times. This is illustrated in Figure 4.8. In this figure, points A, D, E, and G are outside the polygons and points B, D, and F are inside. Note carefully the case of point G; our definition of inside and outside might not be intuitive in some cases.

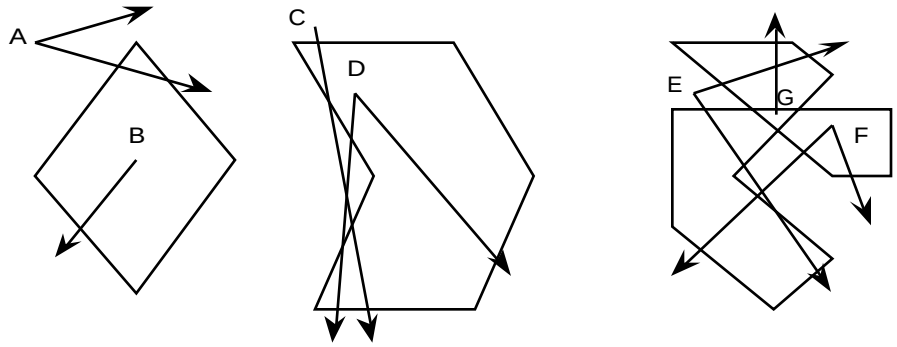


Figure 4.8: Interior and exterior points of a convex polygon (left) and two general polygons (center and right)

Another way to think about convexity is in terms of linear combinations of points. We can define a *convex sum* of points `P0`, `P1`, … `Pn` as a sum $\sum c_i Pi$ where each of the coefficients c_i is non-negative and the sum of the coefficients is exactly 1. If we recall the parametric definition of a line segment, `(1-t)*P0+t*P1`, we note that this is a convex sum. So if a polygon is convex, all convex sums of vertices of the polygon also lie in the polygon, which gives us an alternate definition of convex polygons that can sometimes be useful.

A convex polygon also has a broader property: any point in the polygon is a convex sum of vertices of the polygon. Because this means that the entire interior of the polygon can be expressed as a convex sum of the vertices, we would expect that interpolation processes such as depth (described in an earlier chapter) and color smoothing (described in a later chapter) could be expressed by the same convex sum of these properties for the vertices. Thus convexity is a very important property for geometric objects in computer graphics systems.

As we suggested above, most graphics systems, and certainly OpenGL, require that all polygons be convex in order to render them correctly. If you need to use a polygon that is not convex, you may always subdivide it into triangles or other convex polygons and work with them instead of the original polygon. As an alternative, OpenGL provides a facility to tesselate a polygon—divide it into convex polygons—automatically, but this is a complex operation that we do not cover in these notes.

Polyhedra

As we saw in the earlier chapters, polyhedra are volumes in 3D space that are bounded by polygons. In order to work with a polyhedron you need to define the polygons that form its boundaries. In terms of the scene graph, then, a polyhedron is a group node whose elements are polygons. Most graphics APIs do not provide a rich set of pre-defined polyhedra that you can use in modeling; in OpenGL, for example, you have only the Platonic solids and a few simple polyhedral approximations of other objects (sphere, torus, etc.) A convex polyhedron is one for which any two points in the object are connected by a line segment that is completely contained in the object.

Because polyhedra are almost always defined in terms of polygons, we will not focus on them but will rather focus on polygons. Thus in the next section when we talk about collision detection, the most detailed level of testing will be to identify polygons that intersect.

Collision detection

There are times when we need to know whether two objects meet in order to understand the logic of a particular scene, particularly when that scene involves moving objects. There are several ways to handle collision detection, involving a little extra modeling and several stages of logic, and we outline them here without too much detail because there isn't any one right way to do it.

The first thing you must ask yourself is exactly what kind of collision you want to detect, and what objects in your model could collide. You will see shortly that there is a lot of logic involved in this process, and the two best ways to speed up the process are to avoid making tests when you can, and to make the simplest possible tests when you must test at all.

As we discuss testing below, we will need to know the actual coordinates of various points in 3D world space. You can track the coordinates of a point as you apply the modeling transformation to an object, but this can take a great deal of computation that we would otherwise give to the graphics API, so this works against the approach we have been taking. But your API may have the capability of giving you the world coordinates of a point with a simple inquiry function. In

OpenGL, for example, you can use the function glGetFloatv(GL_MODELVIEW_MATRIX) to get the current value of the modelview matrix at any point; this returns an array of 16 real values that is the matrix to be applied to your model at that point. If you treat this as a 4x4 matrix and multiply it by the coordinates of any vertex, you will get the coordinates of the transformed vertex in 3D eye space. This will give you a consistent space in which to make your tests as described below.

In order to simplify collision detection, it is usual to start thinking of *possible* collisions insteaf of actual collisions. Quick rejection of possible collisions will make a big difference in speeding up handling actual collisions. One standard approach is to use a substitute object instead of the real object, such as a sphere or a box that surrounds the object closely. These are called *bounding objects*, such as bounding spheres or bounding boxes, and they are chosen for the ease of collision testing. It is easy to see if two spheres could collide, because this happens precisely when the distance between their centers is less than the sum of the radii of the spheres. It is also easy to see if two rectangular boxes intersect because in this case, you can test the relative values of the larger and smaller dimensions of each box in each direction. Of course, you must be careful that the bounding objects are defined after all transformations are done for the original object, or you may distort the bounding object and make the tests more difficult.

As you test for collisions, then, you start by testing for collisions between the bounding objects of your original objects. When you find a possible collision, you must then move to more detailed tests based on the actual objects. We will assume that your objects are defined by a polygonal boundary, and in fact we will assume that the boundary is composed of triangles. So the next set of tests are for possible collisons between triangles. Unless you know which triangles in one object are closest to which triangles in another object, you may need to test all possible pairs of triangles, one in each object, so we might start with a quick rejection of triangles.

Just as we could tell when two bounding objects were too far apart to collide, we should be able to tell when a triangle in one object is too far from the bounding object of the other object to collide. If that bounding object is a sphere, you could see whether the coordinates of the triangle's vertices (in world space) are farther from that sphere than the longest side of the triangle, for example, or if you have more detailed information on the triangle such as its circumcenter, you could test for the circumcenter to be farther from the sphere than the radius of the circumcircle. (The circumcenter of a triangle is the common intersection of the three perpendicular bisectors of the sides of the circle; see Figure 4.9 for a sketch that illustrates this.)

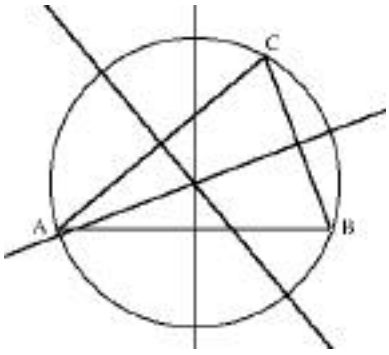


Figure 4.9: the circumcenter and circumcircle of a triangle

After we have ruled out impossible triangle collisions, we must consider the possible intersection of a triangle in one object with a triangle in the other object. In this case we work with each line segment bounding one triangle and with the plane containing the other triangle, and we compute the point where the line meets the plane of the triangle. If the line segment is given by the parametric equation $Q0 + t\ (Q1 - Q0)$ and let the plane of the triangle be $Ax+By+Cz+D=0$, we can

readily calculate the value of *t* that gives the intersection of the line and the plane. If this value of *t* is not between 0 and 1, then the segment does not intersect the plane and we are finished. If the segment does intersect the plane, we need to see if the intersection is within the triangle or not.

Once we know that the line is close enough to have a potential intersection, we move on to test whether the point where the line meets the plane inside the triangle, as shown in Figure 4.10. With the counterclockwise orientation of the triangle, any point on the inside of the triangle is to the left (that is, in a counterclockwise direction) of the oriented edge for each edge of the triangle. The location of this point can be characterized by the cross product of the edge vector and the vector from the vertex to the point; if this cross product has the same orientation as the normal vector to the triangle for each vertex, then the point is inside the triangle. If the intersection of the line segment and the triangle's plane is *Q*, this means that we must have $N \cdot ((P1 - P0) \times (Q - P0)) > 0$ for the first edge, and similar relations for each subsequent edge.

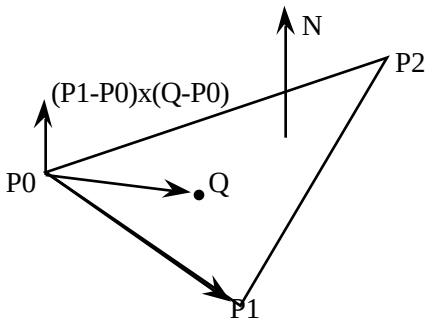


Figure 4.10: a point inside a triangle

Polar, cylindrical, and spherical coordinates

Up to this point we have emphasized Cartesian, or rectangular, coordinates for describing 2D and 3D geometry, but there are times when other kinds of coordinate systems are most useful. The coordinate systems we discuss here are based on angles, not distances, in at least one of their terms. Because graphics APIs generally do not handle non-Cartesian coordinate systems directly, when you want to use them you will need to translate points between these forms and rectangular coordinates.

In 2D space, we can identify any point (X, Y) with the line segment from the origin to that point. This identification allows us to write the point in terms of the angle Θ the line segment makes with the positive X-axis and the distance R of the point from the origin as:

$X = R\cos(\Theta)$, $Y = R\sin(\Theta)$ or, inversely,

$R = sqrt(X^2 + Y^2)$, $\Theta = \arccos(X / R)$

where Θ is the value between 0 and 2π that is in the right quadrant to match the signs of X and Y. This representation (R, Θ) is known as the *polar form* for the point, and the use of the polar form for all points is called the *polar coordinates* for 2D space. This is illustrated in the left-hand image in Figure 4.11.

There are two alternatives to Cartesian coordinates for 3D space. *Cylindrical coordinates* add a third linear dimension to 2D polar coordinates, giving the angle between the X-Z plane and the plane through the Z-axis and the point, along with the distance from the Z-axis and the Z-value of the point. Points in cylindrical coordinates are represented as (R, Θ, Z) with R and Θ as above and

with the *Z*-value as in rectangular coordinates. The right-hand image of Figure 4.11 shows the structure of cylindrical coordinates for 3D space.

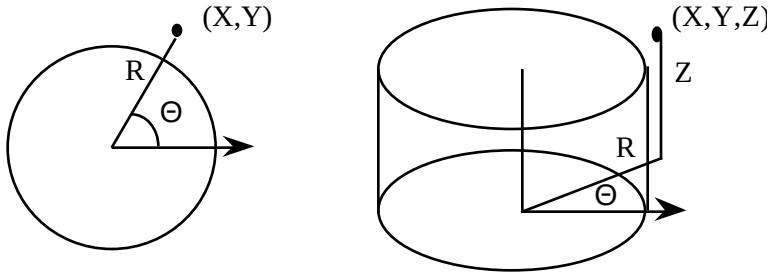


Figure 4.11: polar coordinates (left) and cylindrical coordinates (right)

Cylindrical coordinates are a useful extension of a 2D polar coordinate model to 3D space. They not particularly common in graphics modeling, but can be very helpful when appropriate. For example, if you have a planar object that has to remain upright with respect to a vertical direction, but the object has to rotate to face the viewer in a scene as the viewer moves around, then it would be appropriate to model the object's rotation using cylindrical coordinates. An example of such an object is a billboard, as discussed later in the chapter on high-efficiency graphics techniques.

Spherical coordinates represent 3D points in terms much like the latitude and longitude on the surface of the earth. The latitude of a point is the angle from the equator to the point, and ranges from 90° south to 90° north. The longitude of a point is the angle from the "prime meridian" to the point, where the prime meridian is determined by the half-plane that runs from the center of the earth through the Greenwich Observatory just east of London, England. The latitude and longitude valued uniquely determine any point on the surface of the earth, and any point in space can be represented relative to the earth by determining what point on the earth's surface meets a line from the center of the earth to the point, and then identifying the point by the latitude and longitude of the point on the earth's surface and the distance to the point from the center of the earth. Spherical coordinates are based on the same principle: given a point and a unit sphere centered at that point, with the sphere having a polar axis, determine the coordinates of a point *P* in space by the latitude Φ (angle north or south from the equatorial plane) and longitude Θ (angle from a particular half-plane through the diameter of the sphere perpendicular to the equatorial plane) of the point where the half-line from the center of the sphere, and determine the distance from the center to that point. Then the spherical coordinates of *P* are (R, Θ, Φ).

Spherical coordinates can be very useful when you want to control motion to achieve smooth changes in angles or distances around a point. They can also be useful if you have an object in space that must constantly show the same face to the viewer as the viewer moves around; again, this is another kind of billboard application and will be described later in these notes.

It is straightforward to convert spherical coordinates to 3D Cartesian coordinates. Noting the relationship between spherical and rectangular coordinates shown in Figure 4.9 below, and noting that this figure shows the Z-coordinate as the vertical axis, we see the following conversion equations from polar to rectangular coordinates.

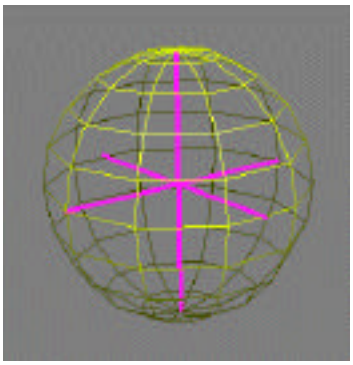
$$x = R\cos(\Phi)\sin(\Theta)$$
$$y = R\cos(\Phi)\cos(\Theta)$$
$$z = R\sin(\Phi)$$

Converting from rectangular to spherical coordinates is not much more difficult. Again referring to Figure 4.9, we see that R is the diagonal of a rectangle and that the angles can be described in terms of the trigonometric functions based on the sides. So we have the equations

$$R = sqrt(X^2 + Y^2 + Z^2)$$
$$\Phi = Arc\sin(Z/R)$$
$$\Theta = \arctan(X/sqrt(X^2 + Y^2))$$

Note that the inverse trigonometric function is the principle value for the longitude (Φ), and the angle for the latitude (Θ) is chosen between 0° and 360° so that the sine and cosine of Θ match the algebraic sign (+ or -) of the X and Y coordinates.

Figure 4.12 shows a sphere showing latitude and longitude lines and containing an inscribed rectangular coordinate system, as well as the figure needed to make the conversion between spherical and rectangular coordinates.



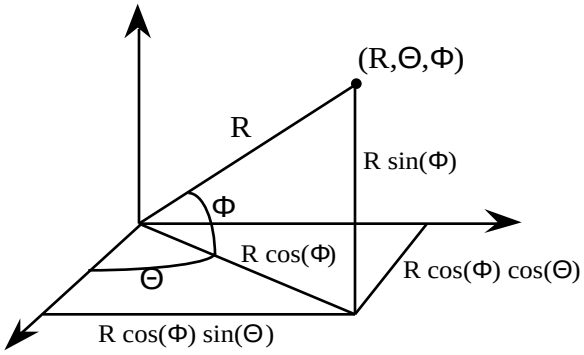


Figure 4.12: spherical coordinates (left);
the conversion from spherical to rectangular coordinates (right)

Higher dimensions?

While our perceptions and experience are limited to three dimensions, there is no such limit to the kind of information we may want to display with our graphics system. Of course, we cannot deal with these higher dimensions directly, so we will have other techniques to display higher-dimensional information. There are some techniques for developing three-dimensional information by projecting or combining higher-dimensional data, and some techniques for adding extra non-spatial information to 3D information in order to represent higher dimensions. We will discuss some ideas for higher-dimensional representations in later chapters in terms of visual communications and science applications.

Chapter 5: Color and Blending

Prerequisites

No background in color is required; this chapter discusses color issues from first principles.

Introduction

Color is a fundamental concept for computer graphics. We need to be able to define colors for our graphics that represent good approximations of real-world colors, and we need to be able to manipulate colors as we develop our applications.

There are many ways to specify colors, but all depend principally on the fact that the human visual system generally responds to colors through the use of three kinds of cells in the retina of the eye. This response is complex and includes both physical and psychological processes, but the fundamental fact of three kinds of stimulus is maintained by all the color models in computer graphics. For most work, the usual model is the RGB (Red, Green, Blue) color model that matches in software the physical design of computer monitors, which are made with a pattern of three kinds of phosphor which emit red, green, and blue light when they are excited by an electron beam. This RGB model is used for color specification in almost all computer graphics APIs, and it is the basis for the discussion here. There are a number of other models of color, and we refer you to textbooks and other sources, especially Foley et al. [FvD], for additional discussions on color models and for more complete information on converting color representations from one model to another.

Because the computer monitor uses three kinds of phosphor, and each phosphor emits light levels based on the energy of the electron beam that is directed at it, a common approach is to specify a color by the level of each of the three primaries. These levels are a proportion of the maximum light energy that is available for that primary, so an RGB color is specified by a triple (r, g, b) where each of the three components represents the amount of that particular component in the color and where the ordering is the red-green-blue that is implicit in the name RGB. This proportion for each primary is represented by a real number between 0.0 and 1.0, inclusive. There are other ways to represent colors, of course. In an integer-based system that is also often used, each color component can be represented by an integer that depends on the color depth available for the system; if you have eight bits of color for each component, which is a common property, the integer values are in the range 0 to 255. The real-number approach is used more commonly in graphics APIs because it is more device-independent. In either case, the number represents the proportion of the available color of that primary hue that is desired for the pixel. Thus the higher the number for a component, the brighter is the light in that color, so with the real-number representation, black is represented by (0.0, 0.0, 0.0) and white by (1.0, 1.0, 1.0). The RGB primaries are represented respectively by red (1.0, 0.0, 0.0), green (0.0, 1.0, 0.0), and blue (0.0, 0.0, 1.0); that is, colors that are fully bright in a single primary component and totally dark in the other primaries. Other colors are a mix of the three primaries as needed.

While we say that the real-number representation for color is more device-independent, most graphics hardware deals with colors using integers. Floating-point values are converted to integers to save space and to speed operations, with the exact representation and storage of the integers depending on the number of bits per color per pixel and on other hardware design issues. This distinction sometimes comes up in considering details of color operations in your API, but is generally something that you can ignore. Some color systems outside the usual graphics APIs use special kinds of color capabilities and there are additional technologies for representing and creating these capabilities. However, the basic concept of using floating-point values for colors is the same as for our APIs. The color-generation process itself is surprisingly complex because the monitor

or other viewing device must generate perceptually-linear values, but most hardware generates color with exponential, not linear, properties. All these color issues are hidden from the API programmer, however, and are managed after being translated from the API representations of the colors, allowing API-based programs to work relatively the same across a wide range of platforms.

In addition to dealing with the color of light, modern graphics systems add a fourth component to the question of color. This fourth component is called "the alpha channel" because that was its original notation [POR], and it represents the opacity of the material that is being modeled. As is the case with color, this is represented by a real number between 0.0 (no opacity — completely transparent) and 1.0 (completely opaque — no transparency). This is used to allow you to create objects that you can see through at some level, and can be a very valuable tool when you want to be able to see more than just the things at the front of a scene. However, transparency is not determined globally by the graphics API; it is determined by compositing the new object with whatever is already present in the Z-buffer. Thus if you want to create an image that contains many levels of transparency, you will need to pay careful attention to the sequence in which you draw your objects, drawing the furthest first in order to get correct attenuation of the colors of background objects.

Definitions

The RGB cube

The RGB color model is associated with a geometric presentation of a color space. That space is a cube consisting of all points (r, g, b) with each of r, g, and b having a value that is a real number between 0 and 1. Because of the easy analogy between color triples and space triples, every point in the unit cube can be easily identified with a RGB triple representation of a color. This gives rise to the notion of the RGB color cube, a 3D space whose coordinates are each bounded by 0 and 1 and with each point associated with a color. The color is, of course, the color whose red, green, and blue values are the *r*, *g*, and *b* coordinates of the point. This identification is very natural and most people in computer graphics think of the color space and the color cube interchangeably.

To illustrate the numeric properties of the RGB color system, we will create the edges of the color cube as shown in Figure 5.1 below, which has been rotated to illustrate the colors more fully. To do this, we create a small cube with a single color, and then draw a number of these cubes around the edge of the geometric unit cube, with each small cube having a color that matches its location. We see the origin (0,0,0) corner, farthest from the viewer, mostly by its absence because of the black background, and the (1,1,1) corner nearest the viewer as white. The three axis directions are the pure red, green, and blue corners. Creating this figure is discussed below in the section on

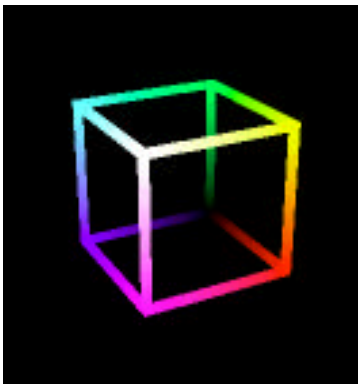


Figure 5.1: tracing the colors of the edges of the RGB cube

creating a model with a full spectrum of colors, and it would be useful to add an interior cube within the figure shown that could be moved around the space interactively and would change color to illustrate the color at its current position in the cube.

This figure suggests the nature of the RGB cube, but a the entire RGB cube is shown from two points of view in Figure 5.2, from the white vertex and from the black vertex, so you can see the full range of colors on the surface of the cube. Note that the three vertices closest to the white vertex are the cyan, magenta, and yellow vertices, while the three vertices closest to the black vertex are the red, green, and blue vertices. This illustrates the additive nature of the RGB color model, with the colors getting lighter as the amounts of the primary colors increase, as well as the subtractive nature of the CMY color model, where the colors get darker as the amounts of color increase. This will be explored later and will be contrasted to the subtractive nature of other color models. Not shown is the center diagonal of the RGB cube from (0, 0, 0) to (1, 1, 1) that corresponds to the colors with equal amounts of each primary; these are the gray colors that provide the neutral backgrounds that are very useful in presenting colorful images.

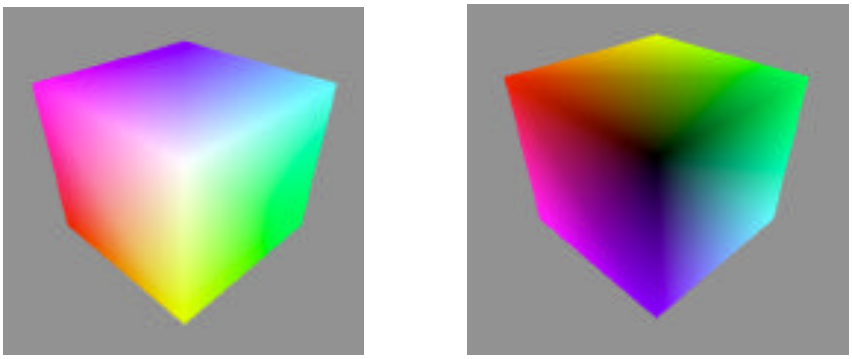


Figure 5.2: two views of the RGB cube — from the white (left) and black (right) corners

Color is ubiquitous in computer graphics, and we can specify color in two ways: by directly setting the color for the objects we are drawing, or by defining properties of object surfaces and lights and having the color generated by a lighting model. In this module we only think about the colors of objects, and save the color of lights and the way light interacts with surfaces for a later module on lighting. In general, the behavior of a scene will reflect both these attributes—if you have a red object and illuminate it with a blue light, your object will seem to be essentially black, because a red object reflects no blue light and the light contains no other color than blue.

Luminance

Luminance of a color is the color's brightness, or the intensity of the light it represents, without regard for its actual color. This concept is particularly meaningful for emissive colors on the screen, because these actually correspond to the amount of light that is emitted from the screen. The concept of luminance is important for several reasons. One is that a number of members of any population have deficiencies in the ability to distinguish different colors, the family of so-called color blindness problems, but are able to distinguish differences in luminance. You need to take luminance into account when you design your displays so that these persons can make sense of them Luminance is also important because part of the interpretation of an image deals with the brightness of its parts, and you need to understand how to be sure that you use colors with the right kind of luminance for your communication. Fore example, in the chapter on visual communication we will see how we can use luminance information to get color scales that are approximately uniform in terms of having the luminance of the color represent the numerical value that the color is to represent.

For RGB images, luminance is quite easy to compute. Of the three primaries, green is the brightest and so contributes most to the luminance of a color. Red is the next brightest, and blue is the least bright. The actual luminance will vary from system to system and even from display device to display device because of differences in the way color numbers are translated into voltages and because of the way the phosphors respond. In general, though, we are relatively accurate if we assume that luminance is calculated by the formula

```
luminance = 0.30*red + 0.59 * green 0.11*blue
```

so the overall brightness ratios are approximately 6:3:1 for green:red:blue.

To see the effects of constant luminance, we can pass a plane $0.30R+0.59G+0.11B+t$ through the RGB color space and examine the plane it exposes in the color cube as the parameter t varies. This is done in Figure 5.3 (to be provided) in both color and grayscale.

Figure 5.3: a plane of constant luminance in the RGB cube in both color (left) and gray (right)

Other color models

There are times when the RGB model is not easy to use. Few of us think of a particular color in terms of the proportions of red, green, and blue that are needed to create it, so there are other ways to think about color that make this more intuitive. And there are some processes for which the RGB approach does not model the reality of color production. So we need to have a wider range of ways to model color to accomodate these realities.

A more intuitive approach to color is found with either of the HSV (Hue-Saturation-Value) or HLS (Hue-Lightness-Saturation) models. These models represent color as a hue (intuitively, a descriptive variation on a standard color such as red, or magenta, or blue, or cyan, or green, or yellow) that is modified by setting its value (a property of darkness or lightness) and its saturation (a property of brightness). This lets us find numerical ways to say "the color should be a dark, vivid reddish-orange" by using a hue that is to the red side of yellow, has a relatively low value, and has a high saturation.

Just as there is a geometric model for RGB color space, there is one for HSV color space: a cone with a flat top, as shown in Figure 5.4 below. The distance around the circle in degrees represents the hue, starting with red at 0, moving to green at 120, and blue at 240. The distance from the vertical axis to the outside edge represents the saturation, or the amount of the primary colors in the particular color. This varies from 0 at the center (no saturation, which makes no real coloring) to 1 at the edge (fully saturated colors). The vertical axis represents the value, from 0 at the bottom (no color, or black) to 1 at the top. So a HSV color is a triple representing a point in or on the cone, and the "dark, vivid reddish-orange" color would be something like (40.0, 1.0, 0.7). Code to display this geometry interactively is discussed at the end of this chapter, and writing an interactive display program gives a much better view of the space.

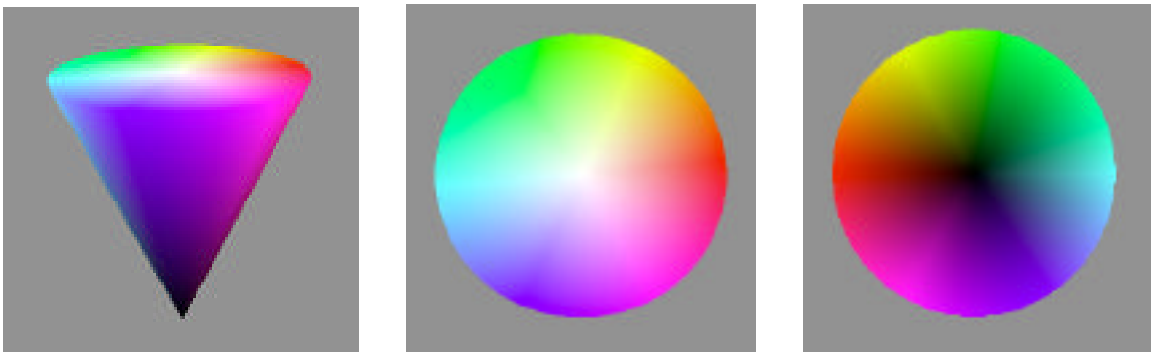


Figure 5.4: three views of HSV color space: side (left), top (middle), bottom (right)

The shape of the HSV model space can be a bit confusing. The top surface represents all the lighter colors based on the primaries, because colors getting lighter have the same behavior as colors getting less saturated. The reason the geometric model tapers to a point at the bottom is that there is no real color variation near black. In this model, the gray colors are the colors with a saturation of 0, which form the vertical center line of the cone. For such colors, the hue is meaningless, but it still must be included.

In the HLS color model, shown in Figure 5.5, the geometry is much the same as the HSV model but the top surface is stretched into a second cone. Hue and saturation have the same meaning as HSV but lightness replaces value, and lightness corresponds to the brightest colors at a value of 0.5. The rationale for the dual cone that tapers to a point at the top as well as the bottom is that as colors get lighter, they lose their distinctions of hue and saturation in a way that is very analogous with the way colors behave as they get darker. In some ways, the HLS model seems to come closer to the way people talk about "tints" and "tones" when they talk about paints, with the strongest colors at lightness 0.5 and becoming lighter (tints) as the lightness is increased towards 1.0, and becoming darker (tones) as the lightness is decreased towards 0.0. Just as in the HSV case above, the grays form the center line of the cone with saturation 0, and the hue is meaningless.

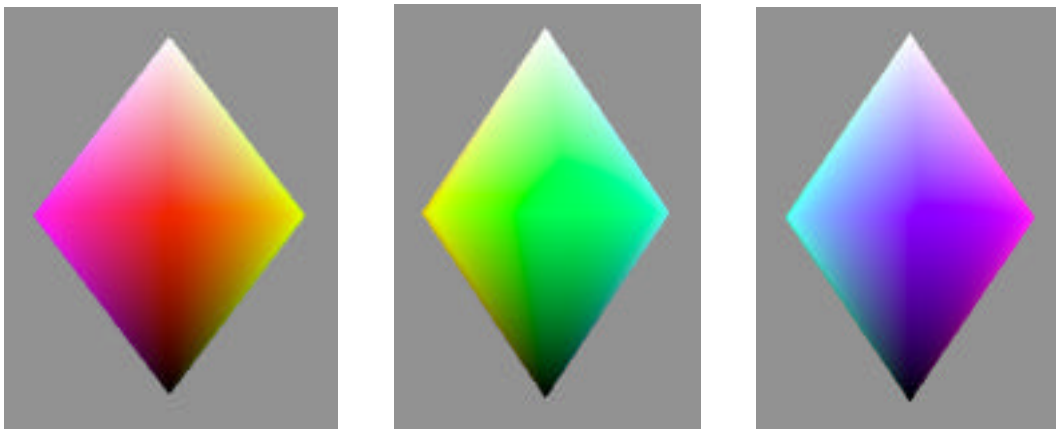


Figure 5.5: the HLS double cone from the red (left), green (middle), and blue(right) directions.

The top and bottom views of the HLS double cone look just like those of the HSV single cone, but the side views of the HLS double cone are quite different. Figure 5.5 shows the HLS double cone from the three primary-color sides: red, green, and blue respectively. The views from the top or bottom are exactly those of the HSV cone and so are now shown here. The images in the figure do

not show the geometric shape very well; the discussion of this model in the code section below will show you how this can be presented, and an interactive program to display this space will allow you to interact with the model and see it more effectively in 3-space.

There are relatively simple functions that convert a color defined in one space into the same color as defined in another space. We do not include all these functions in these notes, but they are covered in [FvD], and the functions to convert HSV to RGB and to convert HLS to RGB are included in the code discussions below about producing these figures.

All the color models above are based on colors presented on a computer monitor or other device where light is emitted to the eye. Such colors are called *emissive* colors, and operate by adding light at different wavelengths as different screen cells emit light. The fact that most color presented by programming comes from a screen makes this the primary way we think about color in computer graphics systems. This is not the only way that color is presented to us, however. When you read these pages in print, and not on a screen, the colors you see are generated by light that is reflected from the paper through the inks on the page. Such colors can be called *transmissive* colors and operate by subtracting colors from the light being reflected from the page. This is a totally different process and needs separate treatment. Figure 5.6 illustrates this principle. The way the RGB add to produce CMY and eventually white shows why emissive colors are sometimes called additive colors, while the way CMY produce RGB and eventually black shows why transmissive colors are sometimes called subtractive colors.

Figure 5.6: emissive colors (left) and transmissive colors (right)

Transmissive color processes use inks or films that transmit only certain colors while filtering out all others. Two examples are the primary inks for printing and the films for theater lights; the primary values for transmissive color are cyan (which transmits both blue and green), magenta (which transmits both blue and red), and yellow (which transmits both red and green). In principle, if you use all three inks or filters (cyan, magenta, and yellow), you should have no light transmitted and so you should see only black. In practice, actual materials are not perfect and allow a little off-color light to pass, so this would produce a dark and muddy gray (the thing that printers call “process black”) so you need to add an extra “real” black to the parts that are intended to be really black. This cyan-magenta-yellow-black model is called CMYK color and is the basis for printing and other transmissive processes. It is used to create color separations that combine to form full-color images as shown in Figure 5.7, which shows a full-color image (left) and the sets of yellow, cyan, black, and magenta separations (right-hand side, clockwise from top left) that are used to create plates to print the color image. We will not consider the CMYK model further in this discussion because its use is in printing and similar technologies, but not in graphics programming. We will meet this approach to color again when we discuss graphics hardcopy, however.

Figure 5.7: color separations for printing

Color depth

The numerical color models we have discussed above are device-independent; they assume that colors are represented by real numbers and thus that there are an infinite number of colors available to be displayed. This is, of course, an incorrect assumption, because computers lack the capability of any kind of infinite storage. Instead, computers use color capabilities based on the amount of memory allocated to holding color information.

The basic model we usually adopt for computer graphics is based on screen displays and can be called direct color. For each pixel on the screen we store the color information directly in the screen buffer. The number of bits of storage we use for each color primary is called the color depth for our graphics. At the time of this writing, it is probably most common to use eight bits of color for each of the R, G, and B primaries, so we often talk about 24-bit color. In fact, it is not uncommon to include the Z-buffer depth in the color discussion, with the model of RGBA color, and if the system uses an 8-bit Z-buffer we might hear that it has 32-bit color. This is not universal, however; some systems use fewer bits to store colors, and not all systems use an equal number of bits for each color, while some systems use more bits per color. The very highest-end professional graphics systems, for example, often use 36-bit or 48-bit color. However, some image formats do not offer the possibility of greater depth; the GIF format, for example specifies 8-bit indexed color in its standard.

One important effect of color depth is that the exact color determined by your color computations will not be displayed on the screen. Instead, the color is aliased by rounding it to a value that can be represented with the color depth of your system. This can lead to serious effects called *Mach bands*, shown in Figure 5.8. These occur because very small differences between adjacent color representations are perceived visually to be significant. Because the human visual system is extremely good at detecting edges, these differences are interpreted as strong edges and disrupt the perception of a smooth image. You should be careful to look for Mach banding in your work, and when you see it, you should try to modify your image to make it less visible. Figure 5.8 shows a small image that contains some Mach bands, most visible in the tan areas toward the front of the image.

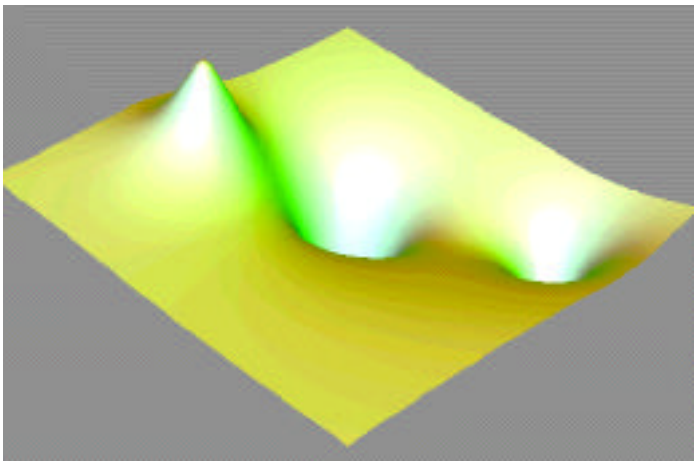


Figure 5.8: an image showing Mach banding

Mach bands happen when you have regions of solid color that differ only slightly, which is not uncommon when you have a limited number of colors. The human eye is exceptionally able to see edges, possibly as an evolutionary step in hunting, and will identify even a small color difference as an edge. These edges will then stand out and will mar the smooth color change that you may want to create in the image.

Color gamut

Color is not only limited by the number of colors that can be displayed, but also by limitations in the technology of display devices. No matter what display technology you use—phosphates on a video-based screen, ink on paper, or LCD cells on a flat panel—there is a limited range of color that the technology can present. This is also true of more traditional color technologies, such as color film. The range of a device is called its *color gamut*, and we must realize that the gamut of our devices limits our ability to represent certain kinds of images. A significant discussion of the color gamut of different devices is beyond the scope of the content we want to include for the first course in computer graphics, but it is important to realize that there are serious limitations on the colors you can produce on most devices.

Color blending with the alpha channel

In most graphics APIs, color can be represented as more than just a RGB triple; it can also include a blending level (sometimes thought of as a transparency level) so that anything with this color will have a color blending property. Thus color is represented by a quadruple (r,g,b,a) and the color model that includes blending is called the RGBA model. The transparency level *a* for a color is called the *alpha value*, and its value is a number between 0.0 and 1.0 that is actually a measure of opacity instead of transparency. That is, if you use standard kinds of blending functions and if the alpha value is 1.0, the color is completely opaque, but in the same situation if the alpha value is 0.0, the color is completely transparent. However, we are using the term "transparent" loosely here, because the real property represented by the alpha channel is *blending*, not transparency. The alpha channel was invented to permit image compositing [POR] in which an image could be laid over another image and have part of the underlying image show through. So while we may say "transparent" we really mean blended.

This difference between blended and transparent colors can be very significant. If we think of transparent colors, we are modeling the logical equivalent of colored glass. This kind of material embodies transmissive, not emissive, colors — only certain wavelengths are passed through,

while the rest are absorbed. But this is not the model that is used for the alpha value; blended colors operate by averaging emissive RGB colors, which is the opposite of the transmissive model implied by transparency. The difference can be important in creating the effects you need in an image. There is an additional issue to blending because averaging colors in RGB space may not result in the intermediate colors you would expect; the RGB color model is one of the worse color models for perceptual blending but we have no real choice in most graphics APIs.

Modeling transparency with blending

Blending creates some significant challenges if we want to create the impression of transparency. To begin, we make the simple observation that is something is intended to seem transparent to some degree, you must be able to see things behind it. This suggests a simple first step: if you are working with objects having their alpha color component less than 1.0, it is useful and probably important to allow the drawing of things that might be behind these objects. To do that, you should draw all solid objects (objects with alpha component equal to 1.0) before drawing the things you want to seem transparent, turn off the depth test while drawing items with blended colors, and turn the depth test back on again after drawing them. This at least allows the possibility that some concept of transparency is allowed.

But it may not be enough to do this, and in fact this attempt at transparency may lead to more confusing images than leaving the depth test intact. Let us consider the case that that you have three objects to draw, and that you will be drawing them in a particular order. For the discussion, let's assume that the objects are numbered 1, 2, and 3, that they have colors C1, C2, and C3, that you draw them in the sequence 1, 2, and 3, that they line up from the eye but have a totally white background behind them, and that each color has alpha = 0.5. Let's assume further that we are not using the depth buffer so that the physical ordering of the objects is not important. The layout is shown in Figure 5.9. And finally, let's further assume that we've specified the blend functions as suggested above, and consider the color that will be drawn to the screen where these objects lie.

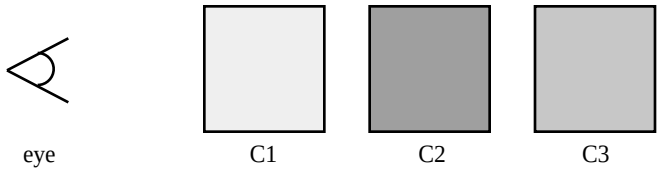


Figure 5.9: the sequence for drawing the objects

When we draw the first object, the frame buffer will have color C1; no other coloring is involved. When we draw the second object on top of the first, the frame buffer will have color `0.5*C1+0.5*C2`, because the foreground (C2) has alpha 0.5 and the background (C1) is included with weight `0.5 = 1-0.5`. Finally, when the third object is drawn on top of the others, the color will be

`0.5*C3+0.5*(0.5*C1+0.5*C2)`, or `0.5*C3+0.25*C2+0.25*C1`.

That is, the color of the most recent object drawn is emphasized much more than the color of the other objects. This shows up clearly in the right-hand part of Figure 5.10 below, where the red square is drawn after the other two squares. On the other hand, if you had drawn object three before object 2, and object 2 before object 1, the color would have been

`0.5*C1+0.25*C2+0.25*C3`,

so the order in which you draw things, not the order in which they are placed in space, determines the color.

But this again emphasizes a difference between blending and transparency. If we were genuinely modeling transparency, it would not make any difference which object were placed first and which

last; each would subtract light in a way that is independent of its order. So this represents another challenge if you would want to create an illusion of transparency with more than one non-solid object.

The problem with the approaches above, and with the results shown in Figure 5.10 below, is that the most recently drawn object is not necessarily the object that is nearest the eye. Our model of blending actually works fairly well if the order of drawing is back-to-front in the scene. If we consider the effect of actual partial transparency, we see that the colors of objects farther away from the eye really are of less importance in the final scene than nearer colors. So if we draw the objects in back-to-front order, our blending will model transparency much better. We will address this with an example later in this chapter.

Indexed color

On some systems, the frame buffer is not large enough to handle three bytes per pixel in the display. This was rather common on systems before the late 1990s and such systems are still supported by graphics APIs. In these systems, we have what is called *indexed color*, where the frame buffer stores a single integer value per pixel, and that value is an index into an array of RGB color values called the *color table*. Typically the integer is simply an unsigned byte and there are 256 colors available to the system, and it is up to the programmer to define the color table the application is to use.

Indexed color creates some difficulties with scenes where there is shading or other uses of a large number of colors. These often require that the scene be generated as RGB values and the colors in the RGB scene are analyzed to find the set of 256 that are most important or are closest to the colors in the full scene. The color table is then built from that analysis.

Besides the extra computational difficulties caused by having to use color table entries instead of actual RGB values in a scene, systems with indexed color are very vulnerable to color aliasing problems. Mach banding is one such color aliasing problem, as are color approximations when pseudocolor is used in scientific applications.

Using color to create 3D images

There is an interesting technique for creating full-color images that your user can view in 3D. It involves the red/blue glasses that are familiar to some of us from the days of 3D movies in the 1950s and that you may sometimes see for 3D comic books or the like. However, most of those were grayscale images, and the technique we will present here works for full-color images.

The images we will describe are called *anaglyphs*. For these we will generate images for both the left and right eyes, and will combine the two images by using the red information from the left-eye image and the blue and green information from the right-eye image as shown in Figure 5.10. The resulting image will look similar to that shown in Figure 5.11, but when viewed through red/blue (or red/green) glasses with the red filter over the left eye, you will see both the 3D content and the colors from the original image. This is a straightforward effect and is relatively simple to generate; we describe how to do that in OpenGL at the end of this chapter.

This description is from the site

http://axon.physik.uni-bremen.de/research/stereo/color_anaglyph/

and the images are copyright by Rolf Henkel. Permission will be sought to use them, or they will be replaced by new work.

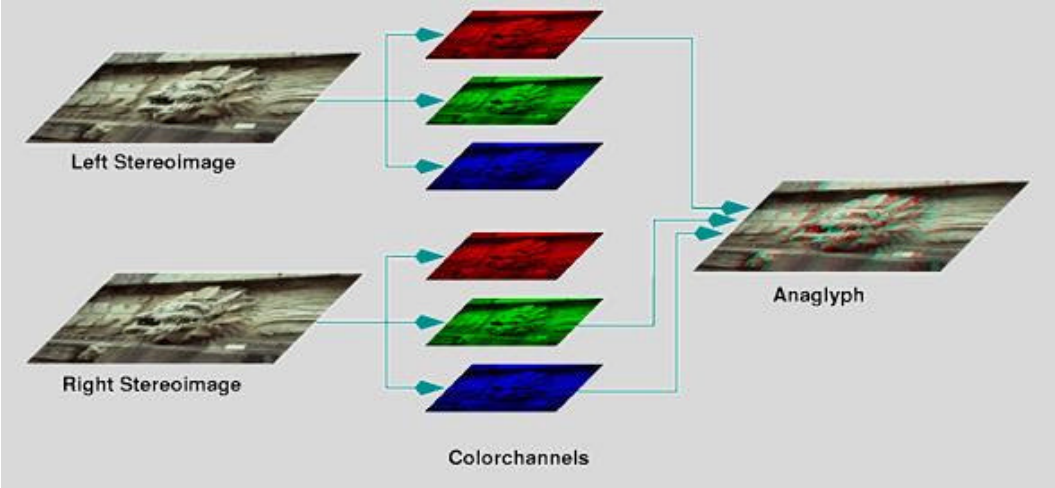


Figure 5.10: creating a blended image from two images

Figure 5.11: an example of a color anaglyph
when viewed with red/blue or red/green glasses, a 3D color image is seen

Some examples

Example: An object with partially transparent faces

If you were to draw a piece of the standard coordinate planes and to use colors with alpha less than 1.0 for the planes, you would be able to see through each coordinate plane to the other planes as though the planes were made of some partially-transparent plastic. We have modeled a set of three squares, each lying in a coordinate plane and centered at the origin, and each defined as having a rather low alpha value of 0.5 so that the other squares are supposed to show through. In this section we consider the effects of a few different drawing options on this view.

The left-hand side of Figure 5.12 below shows the image we get with these colors when we leave the depth test active. Here what you can actually "see through" depends on the order in which you draw the objects. With the depth test enabled, the presence of a transparent object close to your eye prevents the writing of its blend with an object farther away. Because of this, the first coordinate plane you draw is completely opaque to the other planes, even though we specified it as being partly transparent, and a second coordinate plane allows you to see through it to the first plane but is fully opaque to the second plane. We drew the blue plane first, and it is transparent only to the background (that is, it is darker than it would be because the black background shows

through). The green plane was drawn second, and that only allows the blue plane to show through. The red plane was drawn third, and it appears to be fully transparent and show through to both other planes. In the actual working example, you can use keypresses to rotate the planes in space; note that as you do, the squares you see have the same transparency properties in any position.

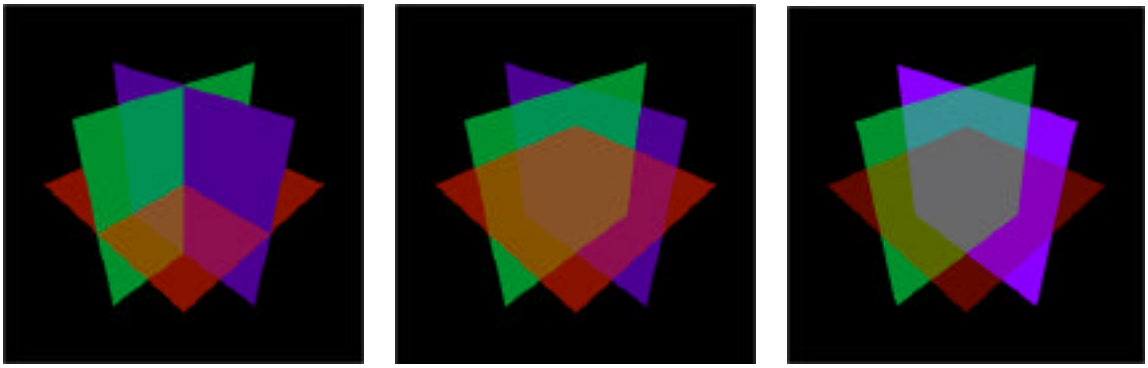


Figure 5.12: the partially transparent coordinates planes (left); the same coordinate planes fully transparent but with same alpha (center); the same coordinate planes with adjusted alpha (right)

However, in the image in the center of Figure 5.12, we have disabled the depth test, and this presents a more problematic situation. In this case, the result is something much more like transparent planes, but the transparency is very confusing because the last plane drawn, the red plane, always seems to be on top because its color is the brightest. This figure shows that the OpenGL attempt at transparency is not necessarily a desirable property; it is quite difficult to get information about the relationship of the planes from this image. Thus one would want to be careful with the images one would create whenever one chose to work with transparent or blended images. This figure is actually created by exactly the same code as the one above with blending disabled instead of enabled.

Finally, we change the alpha values of the three squares to account for the difference between the weights in the final three-color section. Here we use 1.0 for the first color (blue), 0.5 for the second color (green) but only 0.33 for red, and we see that this final image, the right-hand image in Figure 5.12, has the following color weights in its various regions:

- 0.33 for each of the colors in the shared region,
- 0.5 for each of blue and green in the region they share,
- 0.33 each for red and green in the region they share,
- 0.33 for red and 0.67 for blue in the region they share,
- the original alpha values for the regions where there is only one color.

Note that the “original alpha values” gives us a solid blue, a fairly strong green, and a weak read as stand-alone colors. This gives us a closer approximation to the appearance actual transparency for these three colors, with a particular attention to the clear gray in the area they all cover, but there are still some areas that don’t quite work. To get even this close, however, we must analyze the rendering carefully and we still cannot quite get a perfect appearance.

Let’s look at this example again from the point of view of depth-sorting the things we will draw. In this case, the three planes intersect each other and must be subdivided into four pieces each so that there is no overlap. Because there is no overlap of the parts, we can sort them so that the pieces farther from the eye will be drawn first. This allows us to draw in back-to-front order, where the blending provides a better model of how transparency operates. Figure 5.13 shows how this would work. The technique of adjusting your model is not always as easy as this, because it can be difficult to subdivide parts of a figure, but this shows its effectiveness.

There is another issue with depth-first drawing, however. If you are creating a scene that permits the user either to rotate the eye point around your model or to rotate parts of your model, then the model will not always have the same parts closest to the eye. In this case, you will need to use a feature of your graphics API to identify the distance of each part from the eyepoint. This is usually done by rendering a point in each part in the background and getting its Z-value with the current eye point. This is a more advanced operation than we are now ready to discuss, so we refer you to the manuals for your API to see if it is supported and, if so, how it works.

Figure 5.13: the partially transparent planes broken into quadrants and drawn back-to-front

As you examine this figure, note that although each of the three planes has the same alpha value of 0.5, the difference in luminance between the green and blue colors is apparent in the way the plane with the green in front looks different from the plane with the blue (or the red, for that matter) in front. This goes back to the difference in luminance between colors that we discussed earlier in the chapter.

Color in OpenGL

OpenGL uses the RGB and RGBA color models with real-valued components. These colors follow the RGB discussion above very closely, so there is little need for any special comments on color itself in OpenGL. Instead, we will discuss blending in OpenGL and then will give some examples of code that uses color for its effects.

Enabling blending

In order to use colors from the RGBA model, you must specify that you want the blending enabled and you must identify the way the color of the object you are drawing will be blended with the color that has already been defined. This is done with two simple function calls:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The first is a case of the general enabling concept for OpenGL; the system has many possible capabilities and you can select those you want by enabling them. This allows your program to be more efficient by keeping it from having to carry out all the possible operations in the rendering pipeline. The second allows you to specify how you want the color of the object you are drawing to be blended with the color that has already been specified for each pixel. If you use this blending function and your object has an alpha value of 0.7, for example, then the color of a pixel after it has been drawn for your object would be 70% the color of the new object and 30% the color of whatever had been drawn up to that point.

There are many options for the OpenGL blending function. The one above is the most commonly used and simply computes a weighted average of the foreground and background colors, where the

weight is the alpha value of the foreground color. In general, the format for the blending specification is

```
glBlendFunc(src, dest)
```

and there are many symbolic options for the source (src) and destination (dest) blending values; the OpenGL manual covers them all.

A word to the wise...

You must always keep in mind that the alpha value represents a blending proportion, not transparency. This blending is applied by comparing the color of an object with the current color of the image at a pixel, and coloring the pixel by blending the current color and the new color according to one of several rules that you can choose with `glBlendFunc(...)` as noted above. This capability allows you to build up an image by blending the colors of parts in the order in which they are rendered, and we saw that the results can be quite different if objects are received in a different order. Blending does not treat parts as being transparent, and so there are some images where OpenGL simply does not blend colors in the way you expect from the concept.

If you really do want to try to achieve the illusion of full transparency, you are going to have to do some extra work. You will need to be sure that you draw the items in your image starting with the item at the very back and proceeding to the frontmost item. This process was described in the second example above and is sometimes called Z-sorting. It can be very tricky because objects can overlap or the sequence of objects in space can change as you apply various transformations to the scene. You may have to re-structure your modeling in order to make Z-sorting work. In the example above, the squares actually intersect each other and could only be sorted if each were broken down into four separate sub-squares. And even if you can get the objects sorted once, the order would change if you rotated the overall image in 3-space, so you would possibly have to re-sort the objects after each rotation. In other words, this would be difficult.

As always, when you use color you must consider carefully the information it is to convey. Color is critical to convey the relation between a synthetic image and the real thing the image is to portray, of course, but it can be used in many more ways. One of the most important is to convey the value of some property associated with the image itself. As an example, the image can be of some kind of space (such as interstellar space) and the color can be the value of something that occupies that space or happens in that space (such as jets of gas emitted from astronomical objects where the value is the speed or temperature of that gas). Or the image can be a surface such as an airfoil (an airplane wing) and the color can be the air pressure at each point on that airfoil. Color can even be used for displays in a way that carries no meaning in itself but is used to support the presentation, as in the Chromadepth™ display we will discuss below in the texture mapping module. But never use color without understanding the way it will further the message you intend in your image.

Code examples

A model with parts having a full spectrum of colors

The code that draws the edges of the RGB cube uses translation techniques to create a number of small cubes that make up the edges. In this code, we use only a simple cube we defined ourselves (not that it was too difficult!) to draw each cube, setting its color by its location in the space:

```
typedef GLfloat color [4];
color cubecolor;

cubecolor[0] = r; cubecolor[1] = g; cubecolor[2] = b;
cubecolor[3] = 1.0;
glColor4fv(cubecolor);
```

We only use the cube we defined, which is a cube whose sides all have length two and which is centered on the origin. However, we don't change the geometry of the cube before we draw it. Our technique is to use transformations to define the size and location of each of the cubes, with scaling to define the size of the cube and translation to define the position of the cube, as follows:

```
glPushMatrix();
glScalef(scale,scale,scale);
glTranslatef(-SIZE+(float)i*2.0*scale*SIZE,SIZE,SIZE);
cube((float)i/(float)NUMSTEPS,1.0,1.0);
glPopMatrix();
```

Note that we include the transformation stack technique of pushing the current modeling transformation onto the current transformation stack, applying the translation and scaling transformations to the transformation in use, drawing the cube, and then popping the current transformation stack to restore the previous modeling transformation. This was discussed earlier in the chapter on modeling.

The HSV cone

There are two functions of interest here. The first is the conversion from HSV colors to RGB colors; this is taken from [FvD] as indicated, and is based upon a geometric relationship between the cone and the cube, which is much clearer if you look at the cube along a diagonal between two opposite vertices. The second function does the actual drawing of the cone with colors generally defined in HSV and converted to RGB for display, and with color smoothing handling most of the problem of shading the cone. For each vertex, the color of the vertex is specified before the vertex coordinates, allowing smooth shading to give the effect in Figure 5.3. For more on smooth shading, see the later chapter on the topic.

```
void
convertHSV2RGB(float h,float s,float v,float *r,float *g,float *b)
{
// conversion from Foley et.al., fig. 13.34, p. 593
   float f, p, q, t;
   int   k;

   if (s == 0.0) { // achromatic case
      *r = *g = *b = v;
   }
   else {  // chromatic case
      if (h == 360.0) h=0.0;
      h = h/60.0;
      k = (int)h;
      f = h - (float)k;
      p = v * (1.0 - s);
      q = v * (1.0 - (s * f));
      t = v * (1.0 - (s * (1.0 - f)));
      switch (k) {
         case 0: *r = v; *g = t; *b = p; break;
         case 1: *r = q; *g = v; *b = p; break;
         case 2: *r = p; *g = v; *b = t; break;
         case 3: *r = p; *g = q; *b = v; break;
         case 4: *r = t; *g = p; *b = v; break;
         case 5: *r = v; *g = p; *b = q; break;
      }
   }
}
```

```
void HSV(void)
{
#define NSTEPS 36
#define steps (float)NSTEPS
#define TWOPI 6.28318

   int i;
   float r, g, b;

   glBegin(GL_TRIANGLE_FAN); //   cone of the HSV space
      glColor3f(0.0, 0.0, 0.0);
      glVertex3f(0.0, 0.0, -2.0);
      for (i=0; i<=NSTEPS; i++) {
         convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
         glColor3f(r, g, b);
         glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
      }
   glEnd();
   glBegin(GL_TRIANGLE_FAN); //   top plane of the HSV space
      glColor3f(1.0, 1.0, 1.0);
      glVertex3f(0.0, 0.0, 2.0);
      for (i=0; i<=NSTEPS; i++) {
         convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
         glColor3f(r, g, b);
         glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
      }
   glEnd();
}
```

The HLS double cone

The conversion itself takes two functions, while the function to display the double cone is so close to that for the HSV model that we do not include it here. The source of the conversion functions is again Foley et al. This code was used to produce the images in Figure 5.4.

```
void
convertHLS2RGB(float h,float l,float s,float *r,float *g,float *b)
{
// conversion from Foley et.al., Figure 13.37, page 596
   float m1, m2;

   if (l <= 0.5) m2 = l*(1.0+s);
   else          m2 = l + s - l*s;
   m1 = 2.0*l - m2;
   if (s == 0.0) {// achromatic cast
      *r = *g = *b = l;
   }
   else {  // chromatic case
      *r = value(m1, m2, h+120.0);
      *g = value(m1, m2, h);
      *b = value(m1, m2, h-120.0);
   }
}

float value( float n1, float n2, float hue) {
// helper function for the HLS->RGB conversion
```

```
    if (hue > 360.0) hue -= 360.0;
    if (hue < 0.0)   hue += 360.0;
    if (hue < 60.0)  return( n1 + (n2 - n1)*hue/60.0 );
    if (hue < 180.0) return( n2 );
    if (hue < 240.0) return( n1 + (n2 - n1)*(240.0 - hue)/60.0 );
    return( n1 );
}
```

An object with partially transparent faces

The code that draws three squares in space, each centered at the origin and lying within one of the coordinate planes, has a few points that are worth noting. These three squares are colored according to the declaration:

```
GLfloat color0[]={1.0, 0.0, 0.0, 0.5}, // R
       color1[]={0.0, 1.0, 0.0, 0.5}, // G
       color2[]={0.0, 0.0, 1.0, 0.5}; // B
```

These colors are the full red, green, and blue colors with a 0.5 alpha value, so when each square is drawn it uses 50% of the background color and 50% of the square color. You will see that blending in Figure 5.10 for this example.

The geometry for each of the planes is defined as an array of points, each of which is, in turn, an array of real numbers:

```
typedef GLfloat point3[3];
point3 plane0[4]={{-1.0, 0.0, -1.0},   // X-Z plane
                  {-1.0, 0.0,  1.0},
                  { 1.0, 0.0,  1.0},
                  { 1.0, 0.0, -1.0} };
```

As we saw in the example above, the color of each part is specified just as the part is drawn:

```
glColor4fv(color0);       // red
glBegin(GL_QUADS);        // X-Z plane
     glVertex3fv(plane0[0]);
     glVertex3fv(plane0[1]);
     glVertex3fv(plane0[2]);
     glVertex3fv(plane0[3]);
glEnd();
```

This is not necessary if many of the parts had the same color; once a color is specified, it is used for anything that is drawn until the color is changed.

To extend this example to the example of Figure 5.11 with the back-to-front drawing, you need to break each of the three squares into four pieces so that there is no intersection between any of the basic parts of the model. You then have to arrange the drawing of these 12 parts in a sequence so that the parts farther from the eye are drawn first. For a static view this is simple, but as we pointed out in the chapter above on modeling, to do this for a dynamic image with the parts or the eyepoint moving, you will need to do some depth calculations on the fly. In OpenGL, this can be done with the function

```
GLint gluProject(objX,objY,objZ,model,proj,view,winX,winY,winZ)
```

where `objX`, `objY`, and `objZ` are the `GLdouble` coordinates of the point in model space, `model` and `proj` are `const GLdouble *` variables for the current modelview and projection matrices (obtained from `glGetDoublev` calls), `view` is a `const Glint *` variable for the current viewport (obtained from a `glGetIntegerv` call), and `winX`, `winY`, and `winZ` are `GLdouble *` variables that return the coordinates of the point after projection into 3D eye space.

With this information, you can determine the depth of each of the components of your scene and then associate some kind of sequence information based on the depth. This allows you to arrange

to draw them in back-to-front order if you have structured your code so that you can draw your components by techniques such as function calls.

Indexed color

In addition to the RGB and RGBA color we have discussed in this chapter, OpenGL can operate in indexed color mode. However, we will not discuss this here because it introduces few new graphics ideas and is difficult to use to achieve high-quality results. If you have a system that only supports indexed color, please refer to the OpenGL reference material for this information.

Creating anaglyphs in OpenGL

The techniques for creating anaglyphs use more advanced features of OpenGL's color operations. As we saw in Figure 5.10, we need to have both the left-eye and right-eye versions of the image, and we will assume that the images are full RGB color images. We need to extract the red channel of the left-eye image and the green and blue channels of the right-eye image. We will do this by generating both images and saving them to color arrays, and then assembling the parts of the two images into one image by using the red information from the left-eye image and the blue and green information from the right-eye image.

To get these two color arrays for our computer-generated image, we will generate the left-eye image into the back buffer and save that buffer into an array so we can use it later. The back buffer is specified as the buffer for reading with the `glReadBuffer(GL_BACK)` function. You can then load the contents of the buffer into an array with the function

```
glReadPixels(0,0,width,height,GL_RGB,GL_UNSIGNED_BYTE,array)
```

where we assume you are reading the entire window (the lower left corner is (0,0)), your window is `width` pixels wide and `height` pixels high, you are drawing in RGB mode, and your data is to be stored in an array array that contains 3*`width`*`height` unsigned bytes (of type GLUByte). The array needs to be passed as a (`GLvoid *`) parameter. You could use another format for the pixel data, but the `GL_UNSIGNED_BYTE` seems simplest here.

After you have stored the left-eye image, you will do the same with the right-eye image, giving you two arrays of RGB values in memory. Create a third array of the same type, and loop through the pixels, copying the red value from the left-eye pixel array and the green and blue values from the right-eye pixel arrays. You now have an array that merges the colors as in Figure 5.10. You can write that array to the back buffer with the function

```
glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,array)
```

with the parameters as above. This will write the third array back into the frame buffer, where it can be displayed by swapping the buffers.

Questions

1. Look at a digital photograph or scanned image of the real world, and compare it with a grayscale version of itself (for example, you can print the image on a monochrome printer or change your computer monitor to show only grayscale). What parts stand out more in the color image? In the monochrome image?

2. Look at a standard color print with good magnification (a large magnifying glass, a loupe, etc.) and examine the individual dots of each CMYK color. Is there a pattern defined for the individual dots of each color? How do varying amounts of each color make the actual color you see in the image?

3. Look at the controls on your computer monitor, and find ways to displays colors of different depth. (The usual options are 256 colors, 4096 colors, or 16.7 million colors.) Display an image on the screen and change between these color depth options, and observe the differences; draw some conclusions. Pay particular attention to whether you see Mach bands.

Exercises

4. Identify several different colors with the same luminance and create a set of squares next to each other on the screen that are displayed with these colors. Now go to a monochrome environment and see whether the colors of the squares are differentiable.

5. Many computers or computer applications have a "color picker" capability that allows you to choose a color by RGB, HSV, or HLS specifications. Choose one or two specific colors and find the RGB, HSV, and HLS color coordinates for each, and then hand-execute the color conversion functions in this chapter to verify that each of the HSV and HLS colors converts to the correct RGB.

Experiments

6. Draw two or three planar objects on top of each other, with different alpha values in the color of the different objects. Verify the discussion about what alpha values are needed to simulate equal colors from the objects in the final image, and try changing the order of the different objects to see how the alpha values must change to restore the equal color simulation.

Chapter 6: Visual Communication

Prerequisites

Understanding of basic concepts of communication, of visual vocabularies for different kinds of users, and of shaping information with the knowledge of the audience and with certain goals

Introduction

Computer graphics has achieved remarkable things in communicating information to specialists, to informed communities, and to the public at large. This is different from the entertainment areas where computer graphics gets a lot of press because it has the goal of helping the user of the interactive system or the viewer of a well-developed presentation to have a deeper understanding of a complex topic. The range of subjects for this communication include cosmology, in showing how fundamental structures in the universe work; archaeology and anthropology, in showing the ways earlier human groups laid out their structures and cultures; biology and chemistry, in seeing the way electrostatic forces and molecular structures lead to molecular bonding; mathematics, in considering the behavior of highly unstable differential equations; or meteorology, in examining the way global forces such as the temperatures of ocean currents or the depth of the ozone layer affect the weather.

While the importance of visual communication and its associated visual vocabularies has been known by artists, designers, and film directors for a long time, its role in the use of computing in the sciences was highlighted in the 1987 report on Visualization in Scientific Computing [ViSC]. That report noted the importance of computer graphics in engaging the human brain's extraordinary ability to create insight from images. That report noted that Richard Hamming's 1962 quote, "The purpose if computing is insight, not numbers," is particularly applicable when the computing can create images that lead to a deeper and more subtle insight into complex subjects than is possible without images. Indeed, for the student using these notes, we would paraphrase Hamming and say that our purpose for computer graphics is information, not images.

The process of making images—in particular, of making attractive and interesting images with computer graphics using powerful machines and a capable graphics API—is relatively easy. The difficult part of effective computer graphics is the task of understanding your problem and developing ways to present the information that describes your problem so you can make images that communicate with your audience. This short section talks about this task and gives some principles and examples that we hope can start you thinking about this question, but it is a significant task to develop real skill in communicating by means of images. This chapter is relatively early in the overall presentation of graphics primarily to remind you that the main reason we use graphics is to communicate with others, and to help you keep that communication in mind as you learn about making images with computing. Some of the techniques we talk about here will not be covered until later in the notes, but they are not terribly complex, so you should be able to make sense of what the techniques mean even before you have learned how to make them work.

There are several key concepts in the area of communicating effectively through your images. In this chapter we will discuss several techniques and will consider their effect upon communication, but you must realize that this is only an introduction to the topic. Highly-skilled communicators are constantly inventing new ways to present specific information to specific audiences, creating in effect new visual vocabularies for these audiences. We do not try to give you the last answer in visual communication; instead, we are trying to get you to think about the information content of your images and about how you can communicate that to your particular audience. Only a great deal of experience and observation will make you genuinely skilled in this area.

In addition to effective images, visual communication can also include designing interactions so that they give effective support for the visual effects controlled by the interactions. Motion, selection, and control of graphical processes are all reflected in the images presented to the user, so we will discuss some ways you can design the way a user interacts with your programs to support effective and comfortable work.

General issues in visual communication

There are some general points in communicating with your audience that are so important that we want to highlight them here, before we begin looking at the details of communicating with your audience, and discuss some of the issues that are involved in carrying them out.

Use appropriate representation for your information so that your audience will be able to get the most meaning from your images. Sometimes this representation can use color, or sometimes it can use geometry or shapes. Sometimes it will use highly symbolic or synthetic images while sometimes it will use highly naturalistic images. Sometimes it will present the relationships between things instead of the things themselves. Sometimes it will use purely two-dimensional representations, sometimes three-dimensional images but with the third dimension used only for impact, and sometimes three-dimensional images with the third dimension a critical part of the presentation. In fact, there are an enormous number of ways to create representations of information, and the best way to know what works for your audience is probably to observe the way they are used to seeing things and ask them what makes sense for them, probably by showing them many examples of options and alternatives. Do not assume that you can know what they should use, however, because you probably think differently from people in their field and are probably not the one who needs to get the information from the images.

Keep your images focused on just the information that is needed to understand the things you are trying to communicate. In creating the focus you need, remember that simple images create focus by eliminating extraneous or distracting content. Don't create images that are "eye candy" and simply look good; don't create images that suggest relationships or information that are not in the information. For example, when you represent experimental data with geometric figures, use flat shading instead of smooth shading and use only the resolution your data supports because creating higher resolution with smooth interpolation processes, because using smooth shading or smooth interpolation suggests that you know more than your data supports. The fundamental principle is to be very careful not to distort the truth of your information in order to create a more attractive image.

Use appropriate presentation levels for your information. There is a wonderful concept of three levels of information polishing: for yourself (personal), for your colleagues or collaborators (peer), and for an audience when you want to make an impression (presentation). Most of the time when you're trying to understand something yourself, you can use very simple images because you know what you are trying to show with them. When you are sharing your work with your colleagues who have an idea of what you're working on but who don't have the depth of knowledge in the particular problem you're addressing, you might want a bit higher quality to help them see your point, but you don't need to spend a lot of time polishing your work. But when you are creating a public presentation such as a scientific paper or a grant proposal (think of how you would get a point across to a Congressional committee, for example!) you will need to make your work as highly-polished as you can. So your work will sometimes be simple and low-resolution, with very sketchy images; sometimes smoother and with a little thought to how your look at things, perhaps with a little simple animation or with some interaction to let people play with your ideas; and sometimes fully developed, with very smooth animation and high-resolution images, with great care taken to make the maximum impact in the minimum time.

Use appropriate forms for your information. There is a very useful categorization of information (or data) into interval data, ordinal data, and nominal data. Interval data is data that is associated with a meaningful number such as speed, weight, or count. This data is meaningful in individual cases and has a natural representation in real numbers that you can use for your graphical presentation. Ordinal data is data that can be compared with other similar data but that is not necessarily meaningful in itself. Thus the educational level of an individual can be compared with that of another individual, but the most you can usually say with any meaning is that one person has more (or less) education than another. Ordinal data can be represented by size or by color in color ramps that are discussed below. Note that for a color ramp, a viewer can tell which color is higher or lower than another on a color legend but is usually not able to get a careful numeric value from the color. Nominal data is data that describes something with no ordering or numerical meaning; an example might be hair color, with descriptions "red", "brown", "blonde", or "gray". Nominal data can be shown by shapes or individual distinct colors. When you consider your display for a problem, you need to understand what kind of data or information you are working with so you can use an appropriate form for that kind of data.

Be very careful to be accurate with your display. If you have only scattered data for a topic, show only the data you have; do not use techniques such as smooth geometry or smooth coloring to suggest that information is known between the data points. Recognize that a simpler, more accurate representation is going to give your user better information for understanding the situation than a fancier representation. If you use simple numerical techniques, such as difference equations, to model a behavior or to determine the motions in your geometry, say this in a legend or title instead of implying that a more exact solution is presented, so that the modeling you present is not taken as exact. In general, try very hard not to lie with your presentation, whether that lie should be an artifact of your modeling or coding or is an attempt to spin your data to support a particular point of view.

Understand and respect the cultural context of your audience. When you create images to communicate with an audience, that audience can only understand the images in their own

context. This context comes as part of their culture, which might be a professional culture (engineering, medicine, high-energy physics, publishing, education, management ...), a social culture (small-town, major urban, agricultural, rock, ...), a geographical culture (North American, western European, Chinese, Japanese, ...), an ethnic culture (Native American, Zulu, Chicano, ...), or a religious culture (Buddhist, Roman Catholic, fundamentalist Protestant, ...). In each of these you will find some symbols or other visual representations that have a meaning that might be significantly different from the meanings of these same things in other cultural contexts. You must make a careful effort to ensure that your images have the meaning you intend for their audience and not some accidental message caused by your failure to understand the context in which the images are received.

Make your interactions reflect familiar and comfortable relationships between action and effect. If your users are used to a particular kind of controls, simulate those controls and make the actions given by a control mimic the behavior that the control provides in the user's world. The controls you have available are probably familiar from various applications that most people have used, so you can begin by looking at these examples. Some of the controls would naturally be presented in their own control panel; some would be applied in the image itself.

Let's start by looking at some of the control panel vocabulary for interactions. If you want a user to select one among several distinct options in a program, it is natural to make that selection from a radio-button list — a list of the options with a button by each, with each button displaying whether it has been selected, and with a selection on one button canceling any selection on any other button. If you want a user to select zero or more options from a list of options that are not mutually exclusive, then you can use an ordinary button for each, with each button being selected independently of the others. If you want to set a value for a parameter from a continuous range of possible values, then you can use a slider, dial, or thumbwheel to select the value (and you should display the value of the device as it is changed so the user will know what value has been selected). Finally, if you want your user to enter some text (for example, the name of a color if you are using a color naming system instead of RGB values, or the name of a file the program is to use), then you can use a text box to allow text entry. Examples of these controls are displayed in the chapter below on interaction.

If you want the controls to be applied directly in the scene, however, then there is another vocabulary for that kind of interaction. This vocabulary depends on the understanding that the scene represents a certain kind of space, usually a three-dimensional space containing models of physical objects, and that behaviors can be specified in terms of that space. Here we will find object selection by clicking a mouse on a scene and identifying the object that the mouse click has identified, for example. We will also find that we can specify rotations in both latitude and longitude by clicking the mouse in the scene and holding down the button while moving the mouse vertically or horizontally, respectively. We will also find that we can zoom into a scene by specifying that a mouse motion as seen above should be interpreted as a one-dimensional zoom instead of as a two-dimensional rotation. And finally, any of these behaviors could be replaced by using the keyboard to convey the same information, taking advantage of the very large number of degrees of freedom represented by all the keys, of the semantic strength of identifying actions with words and words with their leading characters, and of any familiar key patterns such as the cursor controls from text editors or motion controls from older games.

In all, recognizing that interaction is another form of language and that there are vocabularies from users' backgrounds provides a form of communication that is often visual and that can be built upon to create effective interactive programs.

This chapter makes a number of points about visual communication and computer graphics. These include:

- Shape is a powerful tool that needs to be used carefully to present accurate ideas,
- Color offers many choices that are critical to effective images, and more than anything else, color is how you guide your audience to important parts of your images and convey the exact information you need to give them,
- The choice between naturalistic or artificial shapes and colors is important in communicating appropriate information to your audience,
- There are techniques of color and motion that can convey up to five dimensions of information to your audience, but you need to think carefully about how you encode different kinds of information into each dimension
- An audience may need to see your scene from the right viewpoint and have the right extra information with the image in order to understand the ideas you are presenting,
- Modern graphics APIs and computers have made animated images much simpler to create, and it can be very helpful to take advantage of motion to make your images work better,
- Your images not only can move, but you can create images that allow your audience to interact with the information you are presenting and so can explore the ideas themselves, and
- Your audience will always see your images in the context of their culture, so you must develop an understanding of that culture as part of designing your work.

Together these make up the design issues you must keep in mind as you create your images. They may not all apply in any single piece of work, some take a good bit of work that is outside computer graphics, and many of them take some experience before you can apply them confidently and skillfully. But if you understand their importance, you will create much more effective work.

Shape

Shape is the fundamental part of any image, because with an API-based approach to computer graphics, all our images are built from modeling, and modeling is based on creating geometric objects that have shape. Visual communication begins with images, and thus begins with shapes.

As you saw in the chapter on modeling, there are all sorts of shapes and all sorts of arrangements of those shapes available to you as you create images. You may use simple shapes, emphasizing a basic simplicity in the ideas you are communicating. There is a visual clarity to an image built of simple (or at least apparently simple) shapes; the image will seem uncluttered and it will be easy for your viewer to see what you are presenting.

Shapes are not used arbitrarily, of course. Sometimes your images will describe physical objects, and you will want to create shapes that represent those objects, either by being an accurate version of the objects' shapes or by representing the objects in a recognizable but simplified way. These shapes may be smooth, if you know your theoretical or data values

change smoothly across the display, or they may be coarse and irregular if you have only discrete data and do not know that it is correct to display it smoothly. Sometimes your images will represent concepts that are not embodied in physical objects, and in those cases you will want to create abstract shapes that give your viewer something to represent the ideas you are presenting. These shapes, then, can carry ideas through their position and their properties such as size, shape, color, or motion.

Be careful about the cultural context of shapes you might use as symbols. If you need to use a simple shape to show the position of a data point, for example, you could use a circle, a square, a cross, a pentagram, or a hexagram. A moment's reflection would suggest that the latter three shapes have cultural associations that may or may not be appropriate for your use. In general, be sensitive to this aspect of shapes and recognize that a choice that seems simple to you may have a strong impact on someone else.

For several examples where we will be considering the use of shapes to represent information and comparing that with the use of color, we will use Coulomb's law of electrostatic potential. This law states that at each point of a plane having several point charges, the potential at the point is the sum of the potentials at the point from each of the point charges, and the potential at any point is the point charge divided by the square of the distance between the point and the charge, or:

$$P(x,y) = \sum \frac{Qi}{\sqrt{(x-xi)^2+(y-yi)^2}}$$

where each charge `Qi` is positioned at point `(xi,yi)`. This example is discussed in more detail in the chapter on science applications, but we will consider some representations of the problem as examples of visual presentation options.

If we present the effect of Coulomb's law on a rectangular surface with three fixed point charges, one positive and two negative, we could create the graph of a function of two variables over a domain of the real plane by presenting a surface in three dimensions. In Figure 6.1, we show such a presentation of the function by presenting its 3D surface graph purely as a surface in a fairly traditional way, with an emphasis of the shape of the surface itself. If the emphasis is on the surface itself, this might be a good way to present the graph, because as you can see, the lighting shows the shape well. This is a smooth shape because we are presenting a theory that operates continuously across the space, and the lighting helps to show that shape. The only lie we are telling with this surface is that the surface is bounded, when in fact there are discontinuities at the points where the fixed charges lie because at these points a denominator in the equation is zero.

However, a problem with this kind of representation is that the viewer must understand that there is no actual surface present; instead there is a value that is represented as the height. So this would assume experience in interpreting surfaces as values.

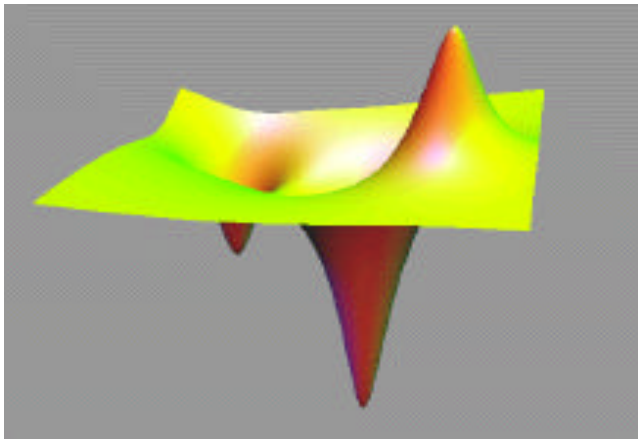


Figure 6.1: traditional surface model presented with three lights to show its shape

Comparing shape and color codings

In the next section of this chapter we will discuss the use of color in communicating with your audience, and will spend some time talking about representing information in terms of color. The three-part Figure 6.2 illustrates three different ways to represent information for a problem in which heat is added to a metal bar at two points and is taken from the bar at another. Here higher and redder values represent higher temperatures, while lower and bluer values represent lower temperatures. The figure shows the state of the temperatures in the bar using both geometric and color encoding (center), using only geometric coding (left), and using only color encoding (right). Note that each of the encodings has its value, but that they emphasize different things. In particular, the height encoding tends to imply that the bar itself actually has a different geometry than a simple rectangular bar, so it might confuse someone who is not used to the substitution of

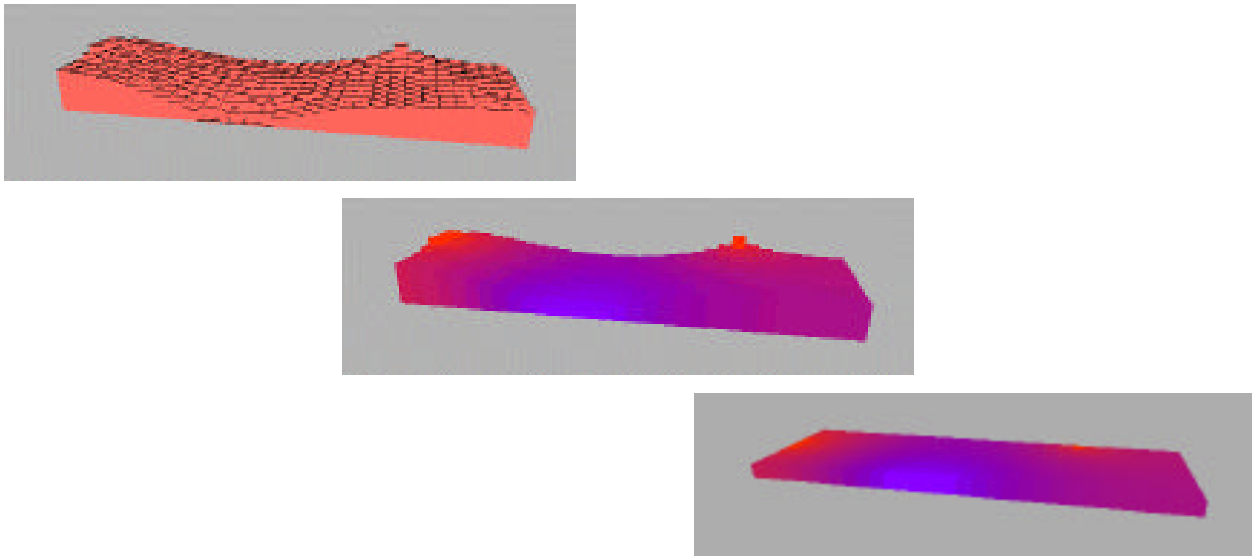


Figure 6.2: three encodings of the same information: temperature in a bar, encoded only through geometry (top left), only through color (bottom right), and through both (center)

geometry for numeric values. The color-only encoding, however, seems easier for a novice to understand because we are used to color coding for heat (think of hot and cold water faucets with different colored inlays) and metal changes color to read when it is sufficiently heated. Thus the way we encode information may depend on the experience of our users and on the conventions they are accustomed to in understanding information. The color encoding in this case follows a traditional blue for cold and red for hot that is common in western culture. Other encodings are found in other groups and could be used here; for example, some branches of engineering commonly use a full rainbow code for colors from magenta for cold, through blue to green and yellow, finally ending with red for hot.

Color

Color is one of the most important tools you have in creating effective communications with computer graphics. It enriches the image and attracts the eye, and it gives you more tools to use in making your points with your audience. As we will see later in this chapter, it can even serve to give you an additional dimension for your images. However, if it is misused, color can work against effective images, so you must be careful about using it; your goal is to create a color scheme that makes apparent sense to the viewer, even if it is completely artificial. In this section we describe many approaches to the question of using appropriate color to help you think about the meaning of color and how you can use it to create strong and effective communication.

Emphasis colors

Your overall image will include whatever information you want to present to the viewer, including context information and various kinds of details that depend on your application. As you design your image, however, you may well want to draw the viewer's attention to specific points in order to show specific things about the content of the display.

There are many ways to draw attention to a specific feature of your scene, but one effective technique is to use a strong, contrasting color for that feature. Such a color will probably be bright and clear, and will be chosen to stand out from the other colors in the overall scene. If you want to have this kind of emphasis, you need to do two things: to design the scene with somewhat muted colors, so that a bright color can stand out from it, and to choose one or more emphasis colors that contrast strongly to the overall colors and are quickly attractive to the eye.

As an example of this, Figure 6.3 shows a surface with a number of control points, and one of the control points is highlighted. Here the background, pool top and bottom, and normal control points are in rather muted colors, but the highlighted control point is in red and can easily be identified as different.

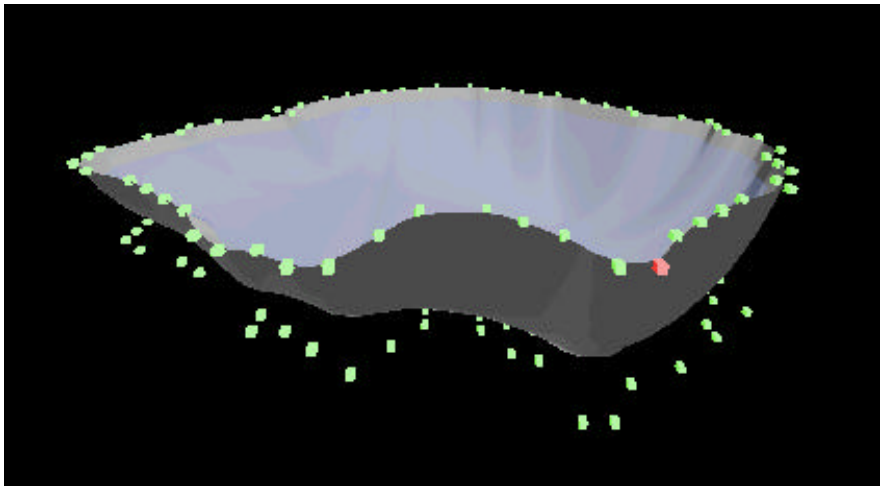


Figure 6.3: an image with one item highlighted for emphasis

Background colors

Images include more than just the objects that are being emphasized. They also include a background color that fills the display space in any area not containing your working image. Background colors should be colors that recede in a user's perception so that the objects to be emphasized stand out against them. In general, a good background color is a dark or neutral color, but black is usually a poor choice because anything that is dark will fade into it. White can be a good background color, because it is certainly neutral although it is not dark. However, just like black, if there are objects that are too light, they may now show up well against white.

If you look at professional photographers and videographers, you will note that they don't use a constant-color background. They use a background with a central highlight that focuses attention on the subject. You should try to use a brighter spot at the center of your background, or possibly a brighter slash of light through the background (usually lower left to upper right) to pull the eye to the center of the image where you'll put your critical content. This idea, and others from photo and video professionals, can help you focus your viewer's attention on the critical part of your images.

Color deficiencies in audience

As you work with color, you must keep in mind that a significant portion of your audience will have difficulties distinguishing certain color combinations. Between 8 and 10 per cent of Caucasian males have a color-vision deficiency; this number is about 4 per cent for non-Caucasian males and only about 0.5 per cent for females. These persons confuse colors, whatever kind of display is used, but most can distinguish colors that differ in luminance, even if they cannot distinguish some differences in chroma. Recall that the luminance of a color is given by the luminance equation:

```
luminance = 0.30*red + 0.59*green + 0.11*blue
```

(where red, green, and blue have the usual convention of lying between zero and one). If your audience will include significant numbers of Caucasian males, you should try to be sure that

elements of your image that your audience needs to distinguish are presented with colors having different luminance, not just different chroma. We will visit this idea further when we talk about color ramps below.

Naturalistic color

If you are working with images of actual objects, you will often want to make those objects seem as realistic as possible to the user. When you do, you have the full capability of your API's modeling, lighting, and shading tools to draw on to create appropriately colored images. You can also apply appropriate texture mapping to give an extra air of realism to your display. These are all discussed in separate chapters in these notes, so we will not discuss them further here.

Pseudocolor and color ramps

Color can mean much more than the actual color of objects. One of the very important uses of color is in serving as a representation of a parameter of the objects being displayed. This parameter could be temperature, velocity, distance, or almost anything you could think of as represented by a number. The value of this numerical parameter can be translated to a color, and then the display of the object in this color carries the information that the object actually has the value represented by the color. This color that represents another value is called a *pseudocolor*.

Pseudocolor separates the concepts of the shape of an object and the colors of the object. It does not try to create a realistic image of the object but applies colors to show some other properties of the object. In Figure 6.4, a house is shown at left in its normal view, and at right with the heat it emits. While this particular pair of images was created with thermal imaging, the principle of showing a property along with the shape is the key to pseudocolor imaging.

Figure 6.4: A house shown in a normal photograph (left) and with thermal imaging (right)

This kind of representation of values by colors is managed by creating *color ramps*, which are one-dimensional sequences of colors with a mapping of any value between 0.0 and 1.0 into a particular color. Color ramps provide the linkage of color to numeric value, so they must be chosen carefully to help the user understand the numeric values the colors display. These ramps may use colors that are customary in the field whose content is being displayed, so there are aspects of cultural context for the ramps. They may be developed to show changes smoothly or to have strong boundaries, depending on the meaning they are to convey. There is something of an art to deciding on the relation between colors and values for a particular application, and you

are encouraged to design your applications so that you can change color ramps easily, allowing you to experiment with different ramps and the different meanings they can convey.

Implementing color ramps

Implementing color ramps is straightforward and is independent of your graphics API. We include some sample code here to show how two color ramps were created so you can adapt these ramps (or similar ones) to your projects. Each assumes that the numerical values have been scaled to this range and returns an array of three numbers that represents the RGB color that corresponds to that value according to the particular representation is uses. This code is independent of the graphics API you are using, so long as the API uses the RGB color model. The first ramp provides color in a rainbow sequence, red-orange-yellow-green-blue-violet. A particularly useful kind of color ramp uses colors that vary uniformly in luminance as the ramp's values range in [0.0, 1.0]. Recalling from the discussion above that luminance is given by

```
luminance = 0.30*red + 0.59*green + 0.11*blue
```

we may use any ramp that is linear in these values. The second ramp has uniform luminance and runs from black through red through yellow to white. Other uniform-luminance color sequences are also possible, of course.

```
void calcRainbow(float yval)
{  if (yval < 0.2)                          // purple to blue ramp
      {myColor[0]=0.5*(1.0-yval/0.2);myColor[1]=0.0;
       myColor[2]=0.5+(0.5*yval/0.2);return;}
   if ((yval >= 0.2) && (yval < 0.40))     // blue to cyan ramp
      {myColor[0]=0.0;myColor[1]=(yval-0.2)*5.0;myColor[2]=1.0;return;}
   if ((yval >= 0.40) && (yval < 0.6))     // cyan to green ramp
      {myColor[0]=0.0;myColor[1]=1.0;myColor[2]=(0.6-yval)*5.0;return;}
   if ((yval >= 0.6) && (yval < 0.8 )      // green to yellow ramp
      {myColor[0]=(yval-0.6)*5.0;myColor[1]=1.0;myColor[2]=0.0;return;}
   if (yval >= 0.8)                         // yellow to red ramp^
      {myColor[0]=1.0;myColor[1]=(1.0-yval)*5.0;myColor[2]=0.0;}
   return;
}

void calcLuminance(float yval)
{  if (yval < 0.30)
      {myColor[0]=yval/0.3;myColor[1]=0.0;myColor[2]=0.0;return;}
   if ((yval>=0.30) && (yval < 0.89))
      {myColor[0]=1.0;myColor[1]=(yval-0.3)/0.59;myColor[2]=0.0;return;}
   if (yval>=0.89)
      {myColor[0]=1.0;myColor[1]=1.0;myColor[2]=(yval-0.89)/0.11;}
   return;
   }
```

The color ramp that is used in the thermal imaging of Figure 6.4 is different from either of these, because in the interval from 0 to 13.9 (degrees Celsius) it runs from black (lowest value) through dark blue to magenta, then to red and on to yellow, and finally to white. This could be implemented by segmenting the range from 0 to about 6 as a ramp from black to blue, then adding red and reducing blue from 6 to 9, then adding yellow to about 12, and then adding blue on up to 13.9. This is left to the student to work out along the lines of the examples above.

Using color ramps

A color ramp represents a one-dimensional space whose values are colors, not numbers. It sets up a relationship between any number between 0.0 and 1.0 and a unique color determined by the way the ramp is coded. As we saw above, the color is used as a surrogate that makes a numerical value visible, so when you want to show an object that has a particular value, you display it in the color that represents that value through the color ramp. You can use this color as an absolute color in a display that does not use lighting, or as a material color in a display that uses lighting; once the color has replaced the numeric value, you can treat the colored model in any way you wish.

At the end of the discussion on shapes, we described an example that showed how shapes or colors could be used to encode information. That example used a simple blue-to-red color change without any explicit use of color ramps. A combination of colors and shapes can also be used with other kinds of examples, so let us consider the Coulomb's law example with the surface shown in Figure 6.1. There the electrostatic potential in a rectangle was shown as a pure surface, but if we consider the use of color ramps to show different values, we get the figures shown in Figure 6.5, with the left-hand image showing the surface with the rainbow color ramp and the right-hand image showing a uniform luminance ramp. You should look at these two images carefully to see whether you can figure out the numeric value of the electrostatic potential at a given point in each. In an actual application, you would not simply display the colored surface but would also include a legend as in Figure 6.4 that identified colors with numeric values; this is discussed below.

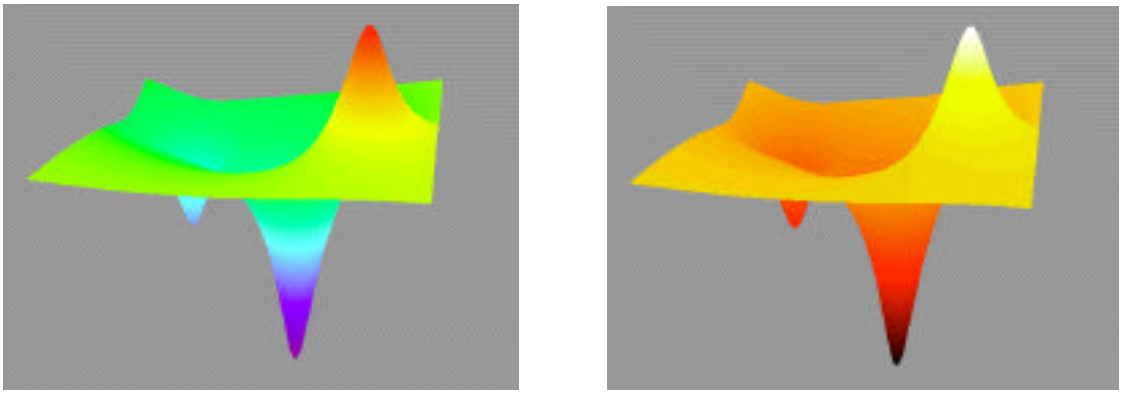


Figure 6.5: electrostatic potential surface model presented with "rainbow" color ramp to emphasize the extreme values (left) and with a uniform luminance distribution of colors (right)

However, perhaps a combination of color and shape such as these might not be the only way to approach the problem. If we think of the surface and the color ramp as different representations of the same space, perhaps it would be useful to think of them as separate displays that are linked in a spatial way. The example of Figure 6.6 below shows one way that might be done, with a lighted surface as one approach and a pseudocolor approach as another, displayed together. Here the rainbow color ramp is used.

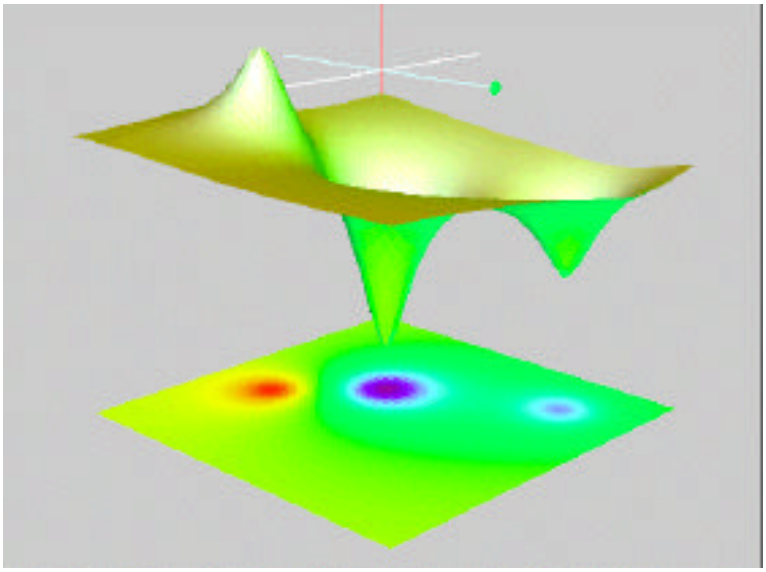


Figure 6.6: a pseudocolor plane with the lighted surface

To light or not to light

If you are developing an image that represents actual objects and uses naturalistic colors, you will almost certainly want to use lighting and shading to make that image as realistic as you can. The representation of actual objects is enhanced by making the objects seem realistic. If the things you display in your scene do not represent actual objects, however, and particularly if your colors are also synthetic and represent some property, then you need to think carefully about whether you should use lighting and shading for your image.

While lighting a scene will make the shapes stand out, there is a danger that issue the colors in the resulting shaded image will not be accurate in representing the values of the data for the scene. If your light definitions for lighting use the same color for ambient and diffuse lighting (especially if that color is white), and if your material color is set by a color ramp that maintains relatively constant luminance and simply changes chroma, there is probably less danger of getting colors that are misinterpreted; the shading provided by the lighting model will change brightness but not color. This is shown by the shaded relief map of South Africa in Figure 6.7, where the colors represent the height (altitude) of the land but the image is shaded as though there were a light low in the eastern sky.

Other instances where lighting might be useful could be where spheres of different sizes and colors were used to represent values, and you could use lighting to emphasize the shape of the spheres.

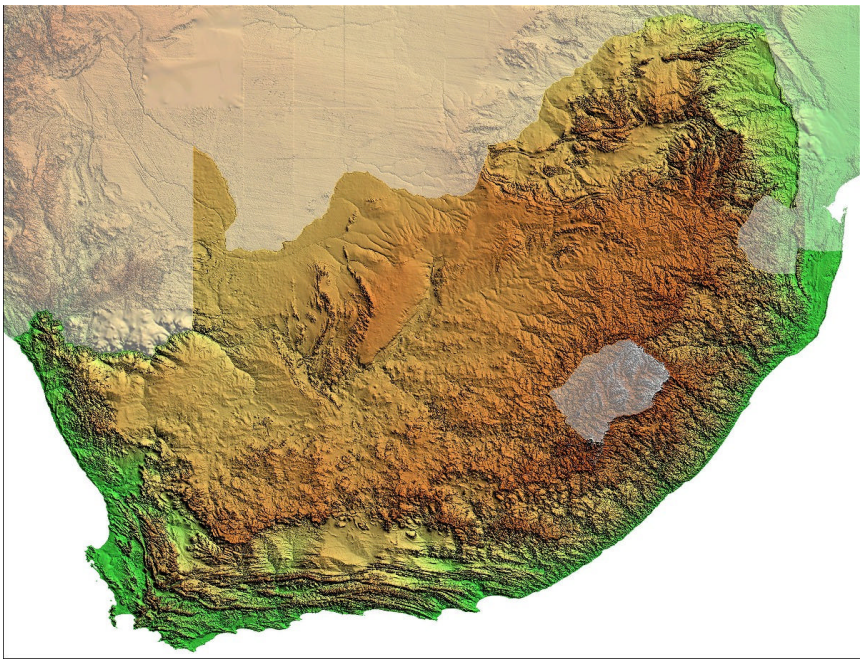


Figure 6.7: a lighted, false-color map of South Africa

Higher dimensions

Surfaces and colorings as described above work well when you are thinking of processes or functions that operate in 2D space. Here you can associate the information at each point with a third dimension and associate that with height or with a color at the point. However, when you get into processes in 3D space, when you think of processes that produce 2D information in 2D space, or when you get into any other areas where you exceed the ability to illustrate information in 3D space, you must find other ways to describe your information.

Perhaps the simplest higher-dimensional situation is to consider a process or function that operates in 3D space and has a simple real value. This could be a process that produces a value at each point in space, such as temperature. There are two simple ways to look at such a situation. The first asks "for what points in space does this function have a constant value?" This leads to what are called *isosurfaces* in the space, and there are complex algorithms for finding isosurfaces or volume data or of functions of three variables. The left-hand part of Figure 6.8 shows a simple approach to the problem, where the space is divided into a number of small cubic cells and the function is evaluated at each vertex on each cell. If the cell has some vertices where the value of the function is larger than the constant value and some vertices where the function is smaller, the continuity of the function assures that the function assumes the constant value somewhere in that cell and a sphere is drawn in each such cell. The second way to look at the situation asks for the values of the function in some 2D subset of the 3D space, typically a plane. For this, we can pass a plane through the 3D space, measure the values of the function in that plane, and plot those values as colors on the plane displayed in space. The right-hand part of Figure 6.8 shows an example of such a plane-in-space display for a function `f(x,y,z)=x*y*z` that is hyperbolic in all three of the `x`, `y`, and `z` components in space. The pseudocolor coding is the uniform luminance ramp described above.

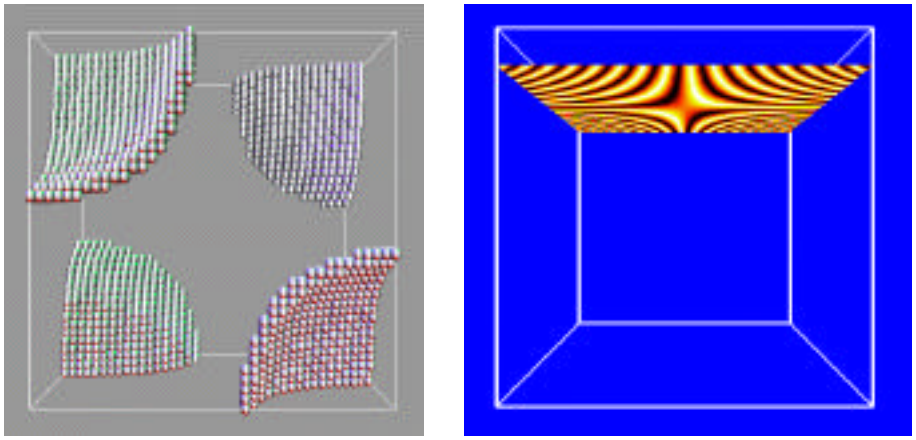


Figure 6.8: a fairly simple isosurface of a function of three variables (left); values of a function in 3D space viewed along a 2D plane in the space (right)

A different approach is to consider functions with a two-dimensional domain and with a two-dimensional range, and to try to find ways to display this information, which is essentially four-dimensional, to your audience. Two examples of this higher-dimension situation are vector-valued functions on a rectangular real space, or complex-valued functions of a single complex variable. Figure 6.9 presents these two examples: a system of two first-order differential equations of two variables (left) and a complex-valued function of a complex variable (right). The domain is the

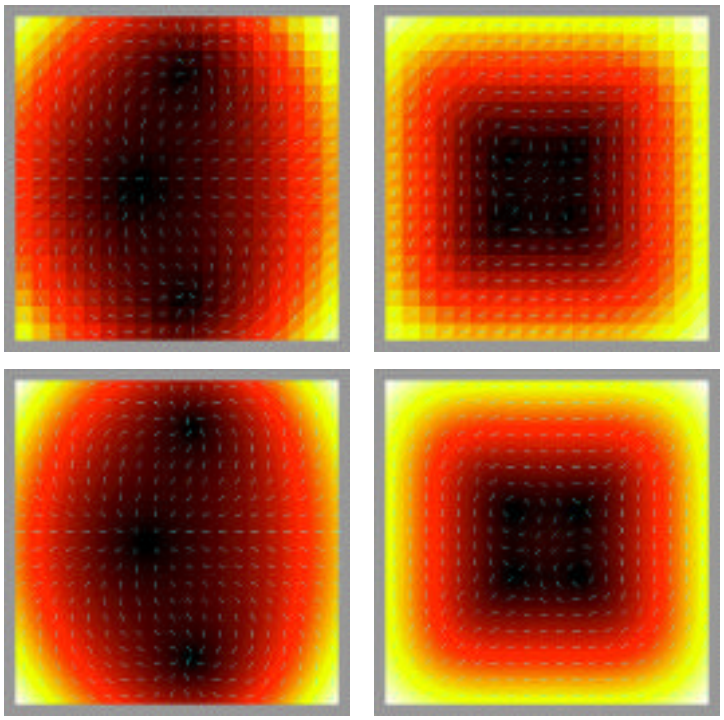


Figure 6.9: two visualizations: a function of a complex variable, left, and a differential equation, right. The top row is low resolution (20x20) and the bottom row is high resolution (200x200)

standard rectangular region of two-dimensional space, and we have taken the approach of encoding the range in two parts based on considering each value as a vector with a length and a direction. We encode the magnitude of the vector or complex number as a pseudocolor with the uniform color ramp as described above, and the direction of the vector or complex number as a fixed-length vector in the appropriate direction. In the top row we use a relatively coarse resolution of the domain space, while in the bottom row we use a much finer resolution. Note that even as we increase the resolution of the mesh on which we evaluate the functions, we keep the resolution of the vector display about the same. This 20x20 vector display mesh is about as fine a resolution as a user can understand on a standard screen.

The displays in Figure 6.9 that combine color and shape are fundamentally 2D images, with the domain of the functions given by the display window and the range of the functions represented by the color of the domain and the direction of the vector. There have been similar visualizations where the range had dimension higher than two, and the technique for these is often to replace the vector by an object having more information [NCSA work reference]. An example is shown in Figure 6.10. Such objects, called *glyphs*, are extremely abstract constructions and need to be designed carefully, because they combine shape and color information in very complex ways. However, they can be effective in carrying a great deal of information, particularly when the entire process being visualized is dynamic and is presented as an animation with the glyphs changing with time.

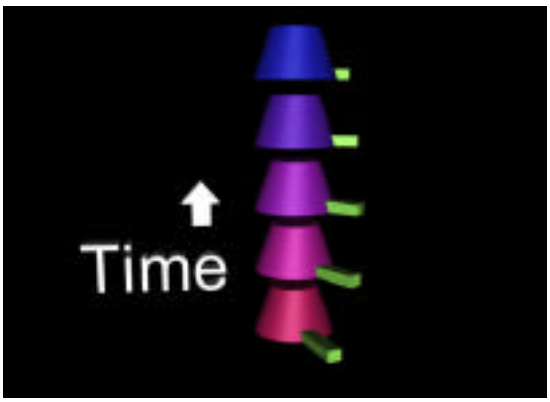


Figure 6.10: glyphs used in an application

Of course, there are other techniques for working with higher-dimensional concepts. One of these is to extend concept of projection. We understand the projection from three-dimensional eye space to two-dimensional viewing space that we associate with standard 3D graphics, but it is possible to think about projections from spaces of four or more dimensions into three-dimensional space, where they can be manipulated in familiar ways. An example of this is the image of Figure 6.11, a image of a hypercube (four-dimensional cube). This particular image comes from an example where the four-dimensional cube is rotating in four-dimensional space and is then projected into three-space.

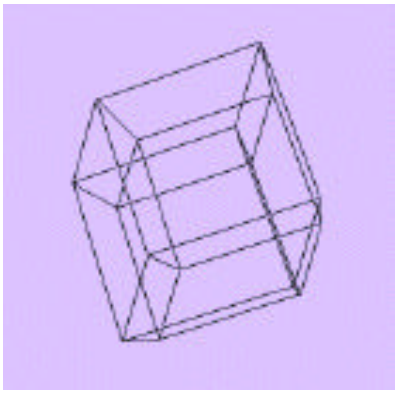


Figure 6.11: a hypercube projected into three-space

Dimensions

We think of computer graphics as being 2D or 3D, because we think of creating images in either a plane or in three-space. But in fact, we have more than three dimensions we can use for images, and understanding this can help us select from options that can make our images more effective.

Three of the dimensions are obvious: the three coordinates of ordinary space. However, we can create motion over time, so we have some control over a fourth dimension of time. And we have seen how we can use color to represent a numeric value, so that allows us to think of color as yet another dimension, our fifth. Let's consider how we might use these dimensions to represent a particular problem: representing the temperature of an object. Recall that early in this chapter we used an example of this type (see Figure 6.2) to compare geometric and color temperature encoding.

There are several contexts for this problem, so let's start with the simplest: the temperature along a wire. In this example, there is one dimension (length) in the physical object, and one more in the temperature at each point. There is no "temperature" dimensions, but we can represent temperature by another dimension: number or color. If we use number, we could create a graph whose horizontal dimension is length and whose vertical dimension is height; this kind of graph is very familiar in textbooks in mathematics and the sciences. If we use color, we have a line that shows different colors at different points on the line, a very different representation but one that looks quite a bit like an actual heated wire. These two representations are shown in Figure 6.12 for a wire that is initially at a uniform heat but has a cold heat sink applied to both ends.

If we want to consider how the temperature on the wire changes over time, we add a third dimension to the question because time is another dimension. Again, we can ask how we would represent that third dimension, and again we have two different answers: as a third numerical dimension, or by animating the original image in time. But since the original image has two choices, we actually have four possible answers for this problem:

1. numerical curve for the temperature at any point in time, extended with numerical values for the third dimension to produce a three-dimensional surface; in this surface, slices parallel to the original curve are the temperatures at any time, while slices perpendicular to the original curve are temperatures at a particular point over time,

2. numerical curve for the temperature at any point in time, extended by animation so the curve changes over time,

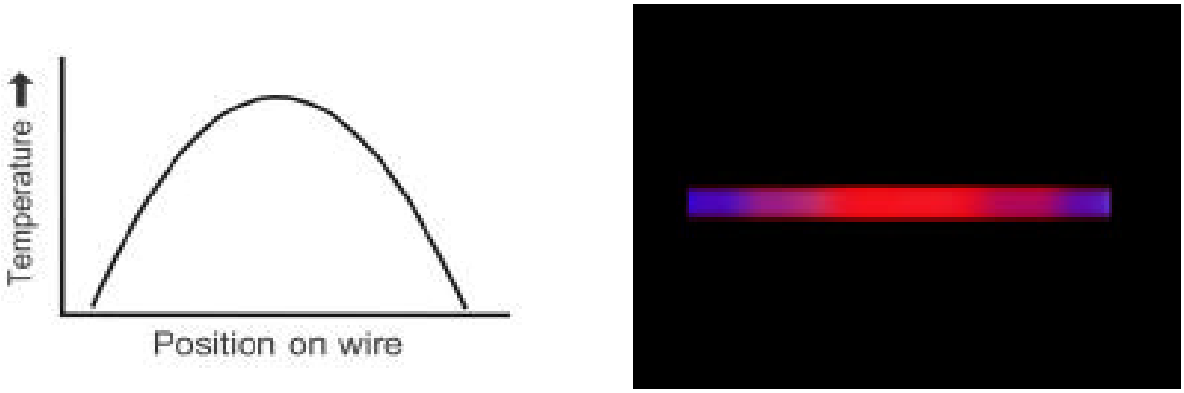


Figure 6.12: two representations of temperature on a wire: curve (left) and color (right)

3. colored line for the temperature at any point in time, extended with numerical values for the third dimension to produce a colored plane; in this plane, slices parallel to the original curve are the temperatures at any time, while slices perpendicular to the original curve are temperatures at a particular point over time,

4. colored line for the temperature at any point in time, extended by animation so the colors change over time.

Figure 6.13 shows two of these representations; unfortunately the media used for these notes does not permit showing animations, so the options that include animation are not shown.

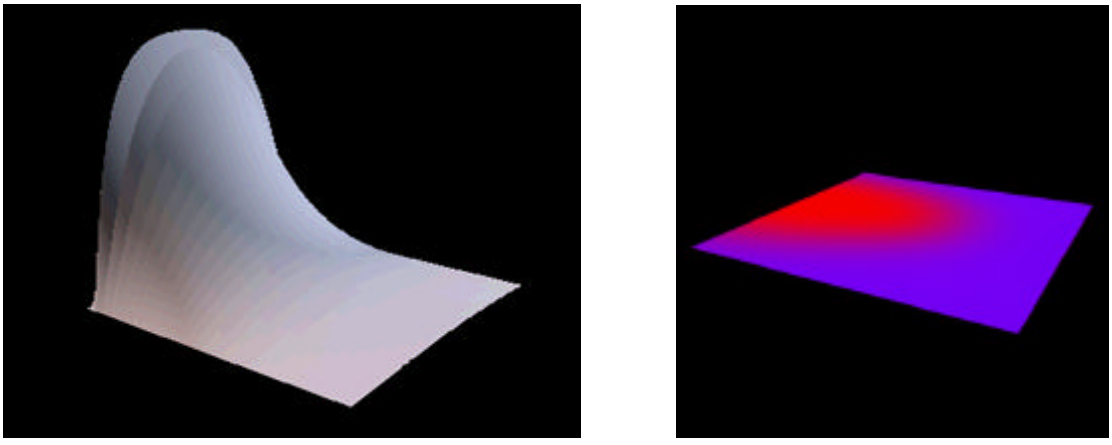


Figure 6.13: change in temperature over time from solutions 1 and 3 above

This problem can be extended to temperatures in a 2D space, and the images in Figure 6.2 show two representations of the problem with the third dimension being temperature: in one the third dimension is height, and in the other the third dimension is color. Note that you can double up on these representations, and in fact the middle image in Figure 6.2 shows a dual encoding of the third dimension. To take this further to consider changes over time, we need a fourth dimension, and time is probably the best one; this would be presented by animating one of the representations from the figure over time.

If we go one step further, we would consider temperature in a 3D space. Here our choices of the fourth dimension are very limited, because time simply does not work differently for different points in the space. Thus we would probably use color as the representation of temperature, our fourth dimension, and we would have to use some kind of higher-dimensional approach to viewing the scene (e.g. slices or equitemperature surfaces). To see how the temperatures would change over time, we would almost certainly animate the display.

So we have many different ways to represent and show various dimensions, and you should look at your options in designing a scene.

Image context

As we discussed when we talked about the context for color, an image is never viewed separately from the rest of the viewer's experience. It is always seen within his or her overall experience and expectations, and must be presented with an understanding of how that experience affects the user's perception. In this section we will talk about a few additional things that help set an image's context for the viewer.

Choosing an appropriate view

When you create a representation of information for an audience, you must focus their attention on the content that you want them to see. If you want them to see some detail in context, you might want to start with a broad image and then zoom into the image to see the detail. If you want them to see how a particular portion of the image works, you might want to have that part fixed in the audience's view while the rest of your model can move around. If you want them to see the entire model from all possible viewpoints, you might want to move the eye around the model, either under user control or through an animated viewpoint. If you want the audience to follow a particular path or object that moves through the model, then you can create a moving viewpoint in the model. If you want them to see internal structure of your model, you can create clipping planes that move through the model and allow the audience to see internal details, or you can vary the way the colors blend to make the areas in front of your structure more and more transparent so the audience can see through them. But you should be very conscious of how your audience will see the images so you can be sure that they see what you need them to see.

Legends to help communicate your encodings

Always be careful to help your audience understand the information you are presenting with your images. Always provide appropriate legends and other textual material to help your audience understand the content of your displays. If you use pseudocolor, present scales that can help a viewer interpret the color information. This allows people to understand the relationships provided by your color information and to understand the context of your problem, and is an important part of the distinction between pretty pictures and genuine information. Creating images without scales or legends is one of the key ways to create misleading visualizations.

The particular example we present here is discussed at more length in the first science applications chapter. It models the spread of a contagious disease through a diffusion process, and our primary interest is the color ramp that is used to represent the numbers. This color ramp is, in fact, the uniform heat ramp introduced earlier in this chapter, with evenly-changing luminance that gets higher (so the colors get lighter) as the values gets higher.

Labels to help communicate your problem

An image alone only makes up part of the idea of using images to present information. Information needs to be put into context to help create real understanding, so we must give our audience a context to help them understand the concept being presented in the image and to see how to decode any use of color or other symbolism we use to represent content. Figure 6.14 shows an image with a label in the main viewport (a note that this image is about the spread of disease) and a legend in a separate viewport to the right of the main display (a note that says what the color means and how to interpret the color as a number). The label puts the image in a general context, and as the results of this simulation (a simulation of the spread of a disease in a geographic region with a barrier) are presented in the main viewport, the legend to the right of the screen helps the viewer understand the meaning of the rising and falling bars in the main figure as the figure is animated and the disease spreads from a single initial infection point.

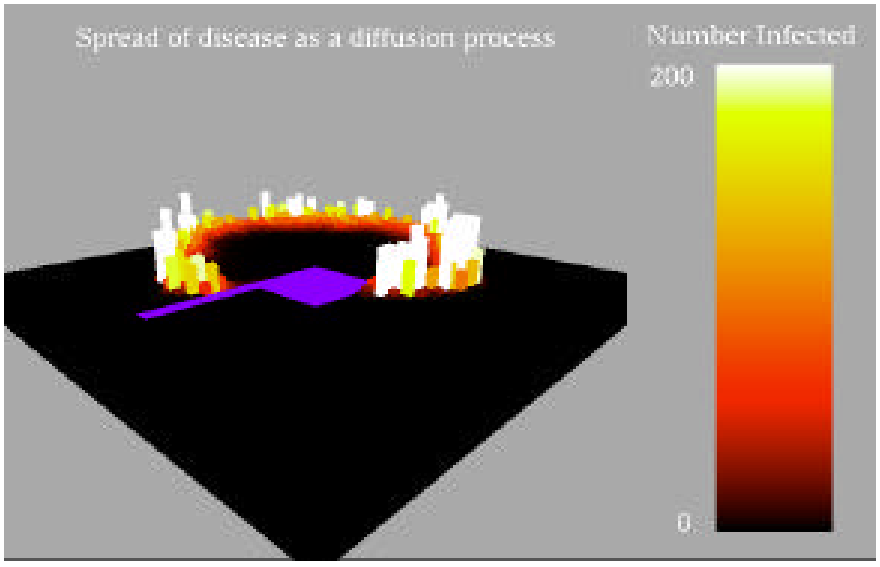


Figure 6.14: an example of figure with a label and a legend to allow the figure to be interpreted

Another form of a label could be some text placed on a billboard that is located in the scene in such a way as to stay out of the important part of the scene. Creating billboards is described in the chapter on texture mapping, but they can give you the effect of floating text in a scene. If this is combined with a line or arrow from the billboard to the particular object that is being described, it can be a very effective way to highlight an object as animation or interaction moves the scene around.

Motion

The ability of a modern graphics API to show motion is a powerful communication tool. Whether the motion is created through animation or through interaction, it allows you to tell a story about your subject that can change through time or can be explored by each member of your audience individually. Presenting a time-changing display is particularly appropriate if you are considering a dynamic situation, which is a common theme in the sciences and in other areas of study. Some phenomena are simply invisible without motion; an example would be detecting a planet among the night stars, where only the different motion of the planet sets it apart to the unassisted eye.

When you create motion, you need to consider exactly what is going to move and the pace at which it is to move. Sometimes you will want to hold most of your scene fixed and have only part of it in motion, which could emphasize the way the moving part changes in relation to the rest of the scene; sometimes you will want to have everything moving so you can describe the overall motion present in the scene. If you are using animation to create this motion, it may be appropriate to use a time parameter in your modeling so that you may simply update the time and redisplay the model. In this case, of course, you will want to be sure that all the parts of your model use time the same way so that the motion is equally paced throughout your scene. If you are using interaction to create the motion, this could get more complicated because you may want to allow your viewer to push some things harder than others, but you again need to create a consistent model of the behavior you will be showing.

The nature of today's computing makes animation an interesting challenge. With faster systems and a growing hardware support for graphics, the time to render a scene keeps decreasing so you get faster and faster frame rates for pure animation. While we wouldn't want to stop or slow this trend, it does mean that we run the risk of creating a real-time online animation that can come to run so fast that it's difficult to understand. Most operating systems have a timed pause function that can be called from your program. You may want to design your animation to have a specific frame rate and use the pause function or a similar system utility to help you maintain the frame rate you want. Of course, this isn't an issue with interactive motion, because the human is always the slowest part of the system and will serve to control his or her own frame rate.

Because sometimes you may want to produce particularly high-quality presentations for the general public or for an audience such as a funding agency, you may need to think about other aspects of a presentation. One of these is certainly sound; in the public presentation world, you will never see an animation without a sound track. Current graphics APIs do not often include sound capability, but we expect that this will come soon and you should think about the sound

that would be used with your work. This could be a recorded voice-over, sound effects that emphasize the motion that is being displayed, or a music track—or all three. If you're going to use video hardcopy for your presentation, you need to consider this now; if you're only going to be doing online work, you need to think about this for the future.

Leaving traces of motion

When you convey information about a moving geometry to your audience, you are likely to use an animation. However, in order that your viewer can see not only the moving parts but also see how these parts have moved, you might want to leave something in the frame to show where the parts were in previous frames. We Might say that you want to show your viewer a *trace* of the motion.

There are two standard ways you can show motion traces. The first is to show some sort of trail of previous positions of your objects. This can be handled rather easily by creating a set of lines or similar geometric objects that show previous positions for each object that is being traced. This trace should have limited length (unless you want to show a global history, which is really a different visualization) and can use techniques such as reduced alpha values to show the history of the object's position. Figure 6.15 shows two examples of such traces; the left-hand image uses a sequence of cylinders connecting the previous positions with the cylinders colored by the object color with reducing alpha values, while the right-hand image shows a simple line trace of a single particle illustrating a random walk situation.

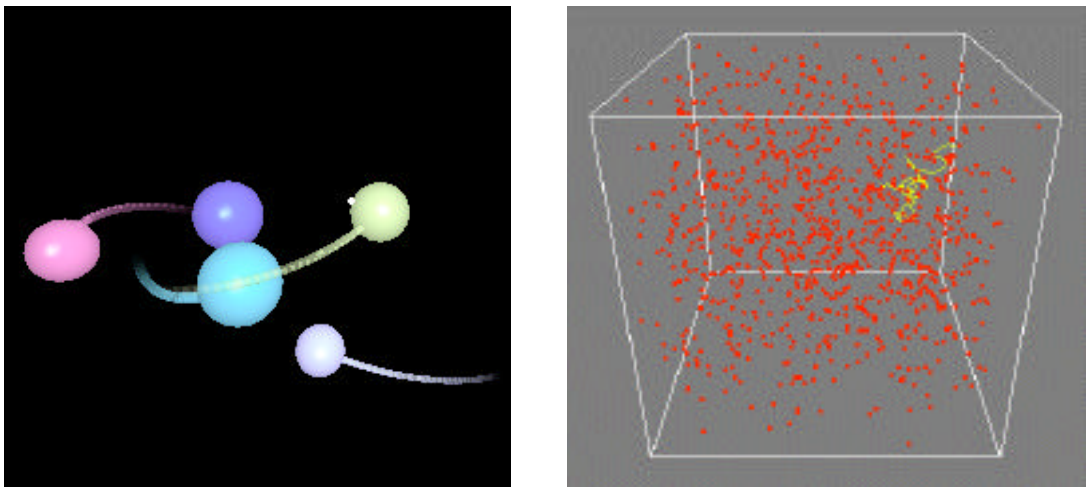


Figure 6.15: two kinds of traces of moving objects

Motion blurring

In many contexts such as sports and high-speed action, we are accustomed to seeing moving objects blurred in a frame. One way to show motion, then, is to create images having a blurred image of those things that are moving, and a crisp image of those things that are in fixed positions. This can be done in several ways, but a standard approach is to use an *accumulation buffer* [ref], a technique that allows you to composite several images of a scene, each taken at a slightly different time. Those objects that are moving will be shown at different positions, so

they will seem blurred; those objects that are fixed will be shown at the same position, so they will be seen as crisp. Many graphics APIs provide the accumulation buffer tool. An example of motion blur is seen in Figure 6.16 below.

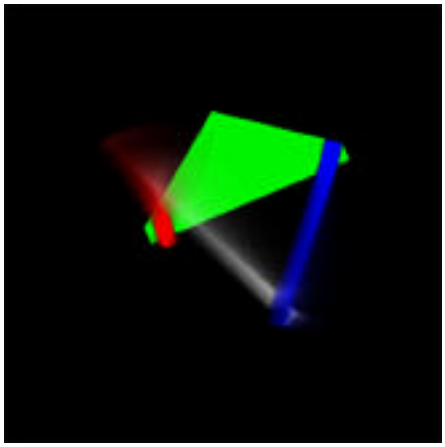


Figure 6.16: a moving mechanism shown with one part fixed and the rest blurred from motion

Interactions

Many graphics applications provide the user the ability to interact with the images they create in order to understand the concepts displayed by the graphics. This interaction is a communication between the user and the program, so it comes under the general theme of this chapter. An interactive application needs to consider the way the user and application communicate about interactions so that the user can make the most productive use of this capability.

One common kind of application is viewing a scene or an object from a number of different viewpoints. When you do this, you are allowing the user to move around the scene and to zoom in or out of the scene. Another way to think of moving around the scene, however, is to rotate the scene in world space while you hold your eye fixed. In either case, you are determining a rotation around a fixed point, either for your eye point or for the scene, with your eye staying the same distance from the scene. This is, in effect, changing the latitude and longitude of the eye point, and it is straightforward to see this as moving in a vertical (latitude) or horizontal (longitude) direction on a sphere that encloses the scene. A natural control for this might be using the mouse in the window, so that when the mouse button is held down, the vertical motion of the mouse is translated into changes in latitude and the horizontal motion of the mouse is translated into changes in longitude. This mouse use is common in applications, and will likely be familiar to an experienced user. The other control for viewing a scene is zooming into or out of the scene, which is a one-dimensional motion. If you have additional buttons on the mouse, you might want to model this with a horizontal mouse motion with a different button pressed, although it might be confusing to have the same action with different buttons treated differently. Another approach would be to use a keyboard action, such as the *f* and *b* keys, move the user forward and back in the scene. This is simple to implement and works well for a single-language application, though it might not work as well if you were to use it for an application that would

be used with languages other than English; however, if you were to allow other key pairs to work in addition to *f* and *b*, it might still work.

Another kind of application involves working with an individual part of an image through selection and manipulation of the part. Here the communication issues are showing that the part may be selected, creating the actual selection, and providing natural ways to manipulate the part. Showing that something may be selected is often done by changing the cursor shape when it is over an item that is selectable, and this can be implemented by using a passive mouse motion and setting a new cursor shape if your graphics API allows it. If the cursor shape cannot be changed, then perhaps the object's display can be subtly different for selectable things than non-selectable things, or perhaps a label or legend can say what is selectable. The actual selection will probably be done with a mouse click, probably with the left button unless there is a good reason for another choice, and the manipulations will be chosen to fit the functions needed. We have described the use of a highlight color to show that something was selected, and above we talked about ways to provide 1D and 2D manipulations. Menus can be used to provide more complex kinds of manipulation, and we would encourage you to think about providing shortcuts in addition to menus to aid the expert user.

While we commonly think of interaction in terms of the common mouse and keyboard inputs, we need to recognize that many of the kinds of interaction we use involve six degrees of freedom (for example, *X-Y-Z* positioning and roll-pitch-yaw motion). There are devices available for high-end graphics systems that provide direct control of six degrees of freedom, and we expect even simple graphics APIs to provide device drivers for these soon. So do not limit your thinking about interaction to the mouse and keyboard, but be prepared to add more sophisticated devices to your toolkit as they become available.

Cultural context of the audience

As we described above, when members of your audience try to understand an image you have created to communicate an idea to them, they will do so within their individual culture. The culture of your audience is a complex issue, involving many different parts—professional cultures, social cultures, geographic cultures, and many others, as we noted above. If someone has to learn how to understand your image (for example, if your image has unique features that the audience has to figure out, or if your figure uses features that mean something different than they would in the audience's culture), then your image will be less effective. You must learn how to express your ideas in the audience's cultural context.

In order to communicate with your audience using images, you must understand the visual vocabularies of your audience's culture. You must understand the nature of the symbols used in the culture, the color schemes familiar to the culture and their meanings, and the way graphic design is used in the culture. For example, while we may be familiar with the spare and open design of traditional Japanese life, the present culture of Japan may be more accurately represented by the crowded, banner-laden Japanese Web sites, which are similar to Japanese newspapers and magazines. If you are reaching a Japanese audience, you will have to choose which of these two approaches you would use in laying out your image.

Most of the work you will do will probably be oriented towards a professional group than a cultural group, however. Thus you will need to understand the use of images to represent concepts in physics, chemistry, biology, or engineering rather than the use of images in Japanese or another ethnic or religious culture. To do this, you will need to know how physicists, chemists, biologists, or engineers are accustomed to using images, and what they may assume your images would mean.

How can you find out what the visual manifestations of a particular culture are? You can research the content of the culture you're reaching by reading the magazines, newspapers, or professional publications of the group. You can develop a bank of images that are commonly found in this audience culture by extracting images from these sources, and you can then test the role of the images in that culture with a panel of experts in the culture—persons deeply familiar with the culture, such as persons from an ethnic or religious culture or professionals from a professional culture. In a similar way, you can develop standard layouts, color schemes, and symbol sets for the culture that have been reviewed and shaped by expert panels. When you are finished, you will have a collection of images, layouts, symbols, and color sets—that is, a design vocabulary—for your target audience and can be comfortable that you have at least the basis for effective communication with that group.

There are references on the meaning of color in various contexts that could be useful to you, especially [THO]. The table below gives one look at this, and the reference contains other ideas about color meanings in context. You are encouraged to look at this further if you need a starting point for cultural issues in color.

Associations to Colors by Professional Group

Color	Process Control Engineers	Financial Managers	Health Care Professionals
Blue	Cold Water	Corporate Reliable	Death
Turquoise (Cyan)	Steam	Cool Subdued	Oxygen deficient
Green	Nominal Safe	Profitable	Infected
Yellow	Caution	Important	Jaundiced
Red	Danger	Unprofitable	Healthy
Purple	Hot Radioactive	Wealthy	Cause for concern

Table 6.17: Associations of Color by Profession

Finally, you should also remember that color is not seen by itself; it is always seen in the context of other colors in the scene, and this context-sensitive color can lead to some surprises. An extensive description of color in context is found in [BRO]. As a very simple example of this context, in Figure 6.18, the image consists of a sequence of black squares, separated by gray lines with a bright white spot at each junction. So why do you only see the white spot you're looking at, when all the other white squares look gray or black?

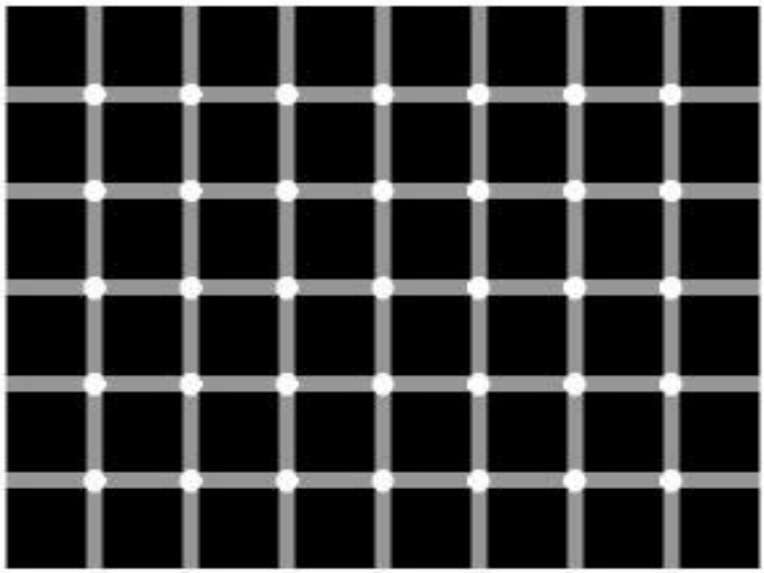


Figure 6.18: a figure that shows artifacts that aren't there

Accuracy

The concept of accuracy is pretty obvious when you are creating images to communicate ideas to your audience. You want to present them with images that represent the information that you have in a way that helps them understand that information clearly and accurately. You must respect your data and not try to go beyond what it can give you. Your goal is not to present pretty pictures; it is to present the data or theory you have as accurately as possible. Of course, making attractive images is useful when you can, even if your goal is accuracy—especially when you are creating images that will be used for public presentation.

Probably the most important point here is to work hard to understand what the data or theory you are working with really tells you, and what it doesn't. For data presentation, the issues are fairly straightforward. If you have data that is organized on a regular mesh, then it is straightforward to use polygons based on the mesh to present that data. If your data is not organized regularly, then you may have to create some sort of representation of the data based on polygons for most graphics APIs to work with. If you do not know *a priori* that there is a smooth variation to the population from which you get your data, then you need to present the data as discrete values and not use smooth interpolations or smooth shading to make your data look smoother. If your data values are spatially distributed and there is no reason to assume anything about the values of the data between your samples, then the best approach might be to display the data in the sampled

space as simple shapes at the sample locations that have different sizes and/or colors to show the values at those positions.

On the other hand, when you display theoretical concepts, you may find that you can calculate the exact theoretical values of your displayed information at some points (or some times), but the behavior between these points or these times is based on operations that cannot readily be solved exactly. This is common, for example, when the theory leads to systems of differential equations that cannot be solved in closed form (that is, whose solutions are not exact expressions). Here you must pay careful attention to the numerical solutions for these operations to be sure that they give you enough accuracy to be meaningful and, in particular, that they do not diverge increasingly from accurate results as your solution points or times move farther from the known values. It can be very useful to have a good knowledge of numerical techniques such as you would get from a study of numerical analysis in order to develop accurate presentations of the theory.

Output media

It is one thing to create images that are effective on the computer screen; it can be quite different to create images that are effective when presented to your audience in other media. As part of your developing your communication, you need to understand what media will be used to get your images to your audience, and work towards those media. Print, online, video, digital video, or physical objects created with rapid prototyping tools all have different properties that need to be considered in making your image presentation effective. We cover most of the issues in output media in the later chapter of these notes on hardcopy, and you need to have a good understanding of media issues, usually by doing a lot of work in different media and studying the properties of the media and how they are used by your audience, before you can be comfortable that you can adjust your work to various media and retain its effectiveness.

Implementing some of these ideas in OpenGL

Many of the techniques we have discussed here are fundamental graphics concepts and have been discussed in other chapters in these notes. Shapes are implemented with the basic modeling tools in OpenGL, colors are implemented with either the absolute color or color through the standard lighting model, and motion is implemented with parametrized transformations and the idle event callback. There is little to be added here to these discussions. Instead, we will focus on a few of the techniques that are not found in these other chapters.

Using color ramps

Because color ramps are used to allow color to represent the numeric value of some parameter, you will use a color ramp by setting the color of some geometric entity (for example, a triangle or quad) based on a calculation of the parameter's value. Using the notion of a ramp developed earlier in this chapter, one would have a code segment such as that below to calculate the RGB components of `myColor[]` from a function `calcRamp(float)` and then set either a material color or an absolute color to the values of `myColor`. This looks like the code below for

absolute colors on a grid of triangles, where the average height of the three vertices for a triangle is used to determine the color.

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<YSIZE-1; j++ ){
      // first triangle in the quad
      glBegin(GL_POLYGON);
         zavg = (height[i][j]+height[i+1][j]+height[i+1][j+1])/3.0;
         calcRamp((zavg-ZMIN)/ZRANGE);
         glColor3f(myColor[0],myColor[1],myColor[2]);
         // now give coordinates of triangle
      glEnd();

      // second triangle in the quad
      glBegin(GL_POLYGON);
         zavg = (height[i][j]+ height[i][j+1]+ height[i+1][j+1])/3.0;
         calcRamp((zavg-ZMIN)/ZRANGE);
         glColor3f(myColor[0],myColor[1],myColor[2]);
         // now give coordinates of triangle
      glEnd();
   }
```

As we said, if you were using a lighted model with materials, you would use the results of the color ramp calculation to set the appropriate material properties for your object.

Legends and labels

Each graphics API will likely have its own ways of handling text, and in this short section we will describe how this can be done in OpenGL. We will also show how to handle the color legend in a separate viewport, which is probably the simplest way to deal with the legend's graphic. This code was used in creating the image in Figure 6.14 above.

The text in the legend is handled by creating a handy function, `doRasterString(...)` that displays bitmapped characters, implemented with the GLUT `glutBitmapCharacter()` function. Note that we choose a 24-point Times Roman bitmapped font, but there are probably other sizes and styles of fonts available to you through your own version of GLUT, so you should check your system for other options.

```
   void doRasterString( float x, float y, float z, char *s)
   {
      char c;

      glRasterPos3f(x,y,z);
      for ( ; (c = *s) != '\0'; s++ )
         glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, c);
   }
```

The rest of the code used to produce this legend is straightforward and is given below. The color of the text is set as an absolute color and lighting, if any, is disabled in order to have complete control over the presentation of the legend. Note that the `sprintf` function in C needs a

character array as its target instead of a character pointer. This code could be part of the display callback function where it would be re-drawn

```
// draw the legend in its own viewport
   glViewport((int)(5.*(float)winwide/7.),0,(int)(2.*(float)winwide/7.),
               winheight);
   glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT);
   ...
   // set viewing parameters for the viewport
   glPushMatrix();
   glEnable (GL_SMOOTH);
   glColor3f(1.,1.,1.);
   doRasterString(0.1, 4.8, 0., "Number Infected");
   sprintf(s,"%5.0f",MAXINFECT/MULTIPLIER);
   doRasterString(0.,4.4,0.,s);
// color is with the heat ramp, with cutoffs at 0.3 and 0.89
   glBegin(GL_QUADS);
      glColor3f(0.,0.,0.);
      glVertex3f(0.7, 0.1, 0.);
      glVertex3f(1.7, 0.1, 0.);
      colorRamp(0.3, &r, &g, &b);
      glColor3f(r,g,b);
      glVertex3f(1.7, 1.36, 0.);
      glVertex3f(0.7, 1.36, 0.);

      glVertex3f(0.7, 1.36, 0.);
      glVertex3f(1.7, 1.36, 0.);
      colorRamp(0.89, &r, &g, &b);
      glColor3f(r,g,b);
      glVertex3f(1.7, 4.105, 0.);
      glVertex3f(0.7, 4.105, 0.);

      glVertex3f(0.7, 4.105, 0.);
      glVertex3f(1.7, 4.105, 0.);
      glColor3f(1.,1.,1.);
      glVertex3f(1.7, 4.6, 0.);
      glVertex3f(0.7, 4.6, 0.);
   glEnd();
   sprintf(s,"%5.0f",0.0);
   doRasterString(.1,.1,0.,s);
   glPopMatrix();
   glDisable(GL_SMOOTH);
// now return to the main window to display the actual model
```

Legends are implemented in much the same way. You can simply create text that is to go into the image however your design dictates, and write that text to the screen with the same kind of text tools.

Creating traces

One way to create the trace of an object, as shown in Figure 6.15 above, is to create a sequence of cylinders that connect a certain number of previous positions of an object and fade out as the points get older. The code below does that, based on a global variable `tails` that maintains the last several positions of the object in an array `list`. Elements of the list describe the previous

positions and directions of the object, as well as the color and length of each segment. The variable `valid` is used as the trace is initialized and not all segments of the trace are yet created.

```
typedef struct {  // hold properties of individual tail cylinders for bodies
   point4 color;
   point3 position;
   point3 direction;
   float length;
   int valid;
} tailstruct;

void draw_tail()
{
   int j;
   float angle;
   point3 rot_vect;
   point3 origin={0.0,0.0,0.0};
   point3 y_point={0.0,1.0,0.0};

   for(j=0; j<T_LENGTH; j++)
      if(tails.list[j].valid) {
         glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,tails.list[j].color);
         // calculate angle to rotate cylinder so it points in right
direction
         angle = asin(tails.list[j].direction[1]
            /sqrt(tails.list[j].direction[0]*tails.list[j].direction[0]
            +tails.list[j].direction[1]*tails.list[j].direction[1]
            +tails.list[j].direction[2]*tails.list[j].direction[2]));
         angle = angle*180/PI+90;
         // calculate vector perpendicular to direction vector and y axis
         // for the line to rotate around.
         normal(tails.list[j].direction, origin, y_point, rot_vect);
         glPushMatrix();
         // move tail segment to right location, rotate, and set length.
         glTranslatef(tails.list[j].position[0],
            tails.list[j].position[1], tails.list[j].position[2]);
         glRotatef(angle, rot_vect[0], rot_vect[1], rot_vect[2]);
         glScalef(1.0, tails.list[j].length, 1.0);
         // draw tail segment as cylinder with 12 slices
         cylinder(radius/30., 12);
         glPopMatrix();
      }
}
```

In the other example of Figure 6.15, showing a random walk of a certain number of steps of a single particle, a similar but simpler kind of process is used because we do not try to fade out the individual steps of the trace. Instead, we merely retain a certain number of previous positions and draw a polyline that connects them in a contrasting color.

Using the accumulation buffer

The accumulation buffer is one of the buffers available in OpenGL to use with your rendering. This buffer holds floating-point values for RGBA colors and corresponds pixel-for-pixel with the frame buffer. The accumulation buffer holds values in the range [-1.0, 1.0], and if any operation

on the buffer results in a value outside this range, its results are undefined (that is, the result may differ from system to system and is not reliable) so you should be careful when you define your operations. It is intended to be used to accumulate the weighted results of a number of display operations and has many applications that are beyond the scope of this chapter; anyone interested in advanced applications should consult the manuals and the literature on advanced OpenGL techniques.

As is the case with other buffers, the accumulation buffer must be chosen when the OpenGL system is initialized, as in

```
glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_ACCUM|GLUT_DEPTH);
```

The accumulation buffer is used with the function `glAccum(mode, value)` that takes one of several possible symbolic constants for its mode, and with a floating-point number as its value. The available modes are

`GL_ACCUM`	Gets RGBA values from the current read buffer (by default the `FRONT` buffer if you are using single buffering or the `BACK` buffer if double buffering, so you will probably not need to choose which buffer to use), converts them from integer to floating-point values, multiplies them by the `value` parameter, and adds the values to the content of the accumulation buffer. If the buffer has bit depth `n`, then the integer conversion is accomplished by dividing each value from the read buffer by 2^n-1.
`GL_LOAD`	Operates similarly to `GL_ACCUM`, except that after the values are obtained from the read buffer, converted to floating point, and multiplied by `value`, they are written to the accumulation buffer, replacing any values already present.
`GL_ADD`	Adds the value of `value` to each of the R, G, B, and A components of each pixel in the accumulation buffer and returns the result to its original location.
`GL_MULT`	Multiplies each of the R, G, B, and A components of each pixel in the buffer by the value of `value` and returns the result to its original location.
`GL_RETURN`	Returns the contents of the accumulation buffer to the read buffer after multiplying each of the RGBA components by `value` and scaling the result back to the appropriate integer value for the read buffer. If the buffer has bit depth `n`, then the scaling is accomplished by multiplying the result by 2^n-1 and clamped to the range [0, 2^n-1].

You will probably not need to use some of these operations to show the motion trace. If we want to accumulate the images of (say) 10 positions, we can draw the scene 10 times and accumulate the results of these multiple renderings with weights 2^{-i} for scene i, where scene 1 corresponds to the most recent position shown and scene 10 to the oldest position. This takes advantage of the fact that the sum

$$\sum_{i=1}^{10} 2^{-i}$$

is very close to 1.0, so we keep the maximum value of the accumulated results below 1.0 and create almost exactly the single-frame image if we have no motion at all. An example of code that accomplishes this is:

```
// we assume that we have a time parameter t for the drawObjects(t)
// function and that we have defined an array times[10] that holds
// the times for which the objects are to be drawn.  This is an example
// of what the manuals call time jittering; another example might be to
// choose a set of random times, but this would not give us the time
// trail we want for this example.
   drawObjects(times[9]);
   glAccum(GL_LOAD, 0.5)
   for (i = 9; i > 0; i--) {
      glAccum(GL_MULT, 0.5);
      drawObjects(times[i-1]);
      glAccum(GL_ACCUM, 0.5);
   }
   glAccum(GL_RETURN, 1.0);
```

The array `times[]` is then updated in the `idle()` function so that each call to the `display()` function shows the object sequence after the next motion step.

A few things to note here are that we save a little time by loading the oldest image into the accumulation buffer instead of clearing the buffer before we draw it, we draw from the oldest to the newest image, we multiply the value of the accumulation buffer by 0.5 before we draw the next image, and we multiply the value of the new image by 0.5 as we accumulate it into the buffer. This accomplishes the successive reduction of the older images automatically.

There are other techniques one could find here, of course. One would be simply to take whatever image you had computed to date, bring it into the accumulation buffer with value 0.5, draw the new scene and accumulate it with weight 0.5, and return the scene with weight 1.0. This would be faster and would likely not show much difference from the approach above, but it does not show the possibilities of drawing a scene with various kinds of jittering, a useful advanced technique.

A word to the wise

Visual communication is much more complex than the material presented in this chapter, which has just tried to hit the high points of the subject to help you get started thinking about the issue. There are full courses in the subject in fields such as communication studies, and there are courses in the arts, cinematography, or videography that focus on just a small portion of this chapter. But if you learn to think about your communication from the beginning of your computer graphics studies, you will find it easier to learn to incorporate more sophisticated and advanced techniques as you develop your professional skills in the field.

Chapter 7: Graphical Problem Solving in Science

Prerequisites:

A knowledge of computer graphics through modeling, viewing, and color, together with enough programming experience to implement the images defined by the science that will be discussed in this section.

Introduction

In the last 20 years, the growing complexity of scientific theory and the volume of scientific data have led to a greatly-increased use of images to represent a wide range of concepts and experiments. The general name for this representation is *scientific visualization*, and it is a critical part of a great deal of scientific work today. The important thing about scientific visualization is not the images that are created to describe scientific principles or processes; it is the graphical problem solving that must be done in order to create the images, the visual communication that must be done to present the images in an effective way, and the understanding of the science that is created by the images for the science community. This understanding can be used to help students learn the science, to help the public or funding sources appreciate the developments in science and support further development, or to help the researchers in the science grasp the implications of their work as they see it more fully. It is not an accident that the general expression of understanding is "Now I see it!" because our mental processes for understanding images are among the most highly developed of all our abilities.

The role of computer graphics in problem solving is to give us a reason to express a problem in visual or geometric terms, to give us a way to take that expression and make it concrete in an actual image, and finally to give us that image as a source of reflection and insight on the problem that can lead to a better understanding of the problem itself. This is described by the closed cycle in Figure 7.1. The initial problem solving task is captured in the *Problem –> Geometry* link below; the concrete expression in computer graphics terms is illustrated by the *Geometry –> Image(s)* link; and the reflection on the problem leading to better understanding is provided by good visual communication in the *Image(s) –> Insight* link. With this approach, supported by the use of computer graphics, we are able to reach the *Insight –> Problem* link where the real advancement of our understanding is provided. Together these components describe a powerful way to address any kind of problem by engaging a combination of analytic and visual capabilities.

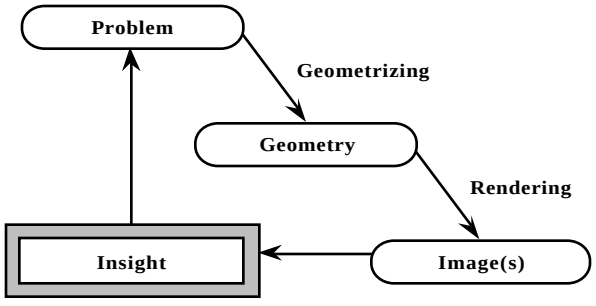


Figure 7.1: The graphical problem-solving cycle

As we consider our problems and how we can view them graphically or geometrically, we find that we have some basic questions to answer. Can our problem naturally be expressed in terms of objects that have a natural image (automobiles, houses, ...) or have a familiar representation (bar, sphere, pyramid, ...)? If so, then you can start with these natural or familiar objects and see how you can represent your problem in their terms. If not, then you need to try to find or create such a

representation of the problem, because you cannot display a solution without geometry. As Galileo said,

> "Philosophy is written in this grand book the universe, which stands continually open to our gaze. But the book cannot be understood unless one first learns to comprehend the language and to read the alphabet in which it is composed. It is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures, without which it is humanly impossible to understand a single word of it; without these, one wanders about in a dark labyrinth." [Galileo's Daughter, p. 16]

The overall process we describe here is what we call graphical problem solving. The key point is being able to identify ways to describe a problem in geometric terms that permit the design and creation of images that represent part of the problem. The general process or representing a problem in ways that permit better understanding is called *modeling* the problem, and modeling is a large topic that is treated in complete books of its own. However, we want to help you understand the kind of modeling that is part of graphical problem solving, and in this chapter we will describe some kinds of graphical modeling that have been used in addressing various problems in science. These will be relatively simple models because we're looking at the kind of graphical models that can be created using your beginning graphics tools; much more complex and sophisticated kinds of models have been created and are being used, and you should look at the general literature of scientific visualization to get an understanding of the greater depth that is possible.

As we look at the examples of graphical modeling, we will examine a number of different kinds of scientific problems and for each we will describe the problem and how it can be modeled graphically, and where appropriate describe tradeoffs that we made in choosing the particular representation, and we will describe the way we build an image that is based on that model. Later in the chapter there will be a discussion of some details in the computer modeling (as distinct from the problem modeling) that would go into creating the images for the models; these will sometimes include discussions of the graphics techniques but will sometimes focus on other programming issues. The end product of this work will be a description of the process of problem solving that you should be able to take to your own projects or problems. Together with the set of examples where this process is discussed, this should give you the tools to use graphical problem solving to accomplish your own goals.

Before we move to discussing the modeling examples, we should say a word about the kinds of data that you may encounter when you're working with scientific information. There are different kinds of data, called interval, ordinal, and nominal data, that must be treated differently when you represent them graphically. Interval data can be represented by real numbers, ordinal data are data that have a natural ordering but no meaningful numerical representation, and nominal data are data that are in different categories but have no ordering. There are few challenges in dealing with interval data, because we are familiar with numerical values and their representations as values in dimensions or even as color ramps. With ordinal data, we can use positioning, relative sizes, or colors of increasing brightness or other property. With nominal data, we can use different shapes or distinct colors, but we probably need to use some sort of legend to identify which representation we use for each value because there will likely be no natural identification available.

When we work with interval data of high dimension and the dimension of the data, or the dimensions of the domain and range of a function together, exceeds the three dimensions we can directly plot, we are likely to need to allow the user some options in viewing the data space. Data exploration can be supported by providing various projections and controls for moving around the data to see the different projections. We are used to thinking of 2D screen projections of 3D space, but we will need to consider projections into 3D space from higher-dimension spaces as well. With the growth of genuine immersive viewing devices for 3D viewing, it may not be necessary to project below 3D space at all in the future.

Examples

In this chapter we will describe a number of techniques you can use to work on problems in the sciences. In a sense, presenting a set of techniques may tend to make you focus on the graphical technique rather than the problem you are examining, so you may miss the point of starting with the problem first and finding a graphical or visual representation later. However, our goal is to give you a set of arrows from which you can choose when you shoot at a problem. Learning to analyze the problem and find an appropriate technique will only come with practice.

Our techniques are taken from several years of looking at discussions of visual approaches to the sciences. They are usually not the most sophisticated kinds of images because we are considering work that you can do based on your own programming with a simple graphics API such as OpenGL, rather than with sophisticated scientific visualization tools. If you master these simpler techniques, however, you should have a very good background to understand the way the more complex tools work and to take advantage of them when they are available.

Diffusion

Diffusion is a process where a property that is present at one point in a material is spread throughout the material by migrating from its original point to adjacent points over time. When this is modeled for computational purposes, the material is usually divided into a grid of "points" that are usually a unit of area or volume, and the amount of the property at each grid point is given some numeric value. The property might be a quantity of salt dissolved in a unit volume of water, a quantity of heat in a unit volume of a material, or the number of events in a unit area. The process is modeled by assuming that quantities of the property transfer from a given grid point to neighboring grid points proportionately to the amount of property present, either determinately or with some randomness included in the process. This is a very general kind of process that can be applied to a wide range of problems, and in this section we will look at two models built from it.

Temperatures in a bar

Let us start with a rectangular bar of some material that is embedded in an insulating medium and has points at which fixed-temperature connectors may be attached. Our goal is to consider the distribution of temperatures throughout the bar over time. We assume that the bar has constant thickness and that the material in the bar is homogeneous throughout its thickness at any point, so we may treat the bar as a 2D entity. The bar may be homogeneous or heterogeneous; the material the bar is made of may have varying thermal conductivity properties; the connectors may be connected or disconnected, and may have time-varying temperatures (but their temperatures are directed by an outside agent and have no relation to the temperatures of the bar). The basic property of the distribution of heat is governed by the *heat equation* $\frac{\partial F}{\partial t} = k\frac{\partial^2 F}{\partial x^2}$, the partial differential equation that describes heat transfer, with the constant ***k*** determined by the material in the bar. This equation says, basically, that at a given point, the rate of change of heat with time is proportional to the gradient (the second derivative) of the heat with space—that is, that the change in heat with time is associated with changes in the heat transfer over space. If the distribution of heat in space, $\frac{dF}{dx}$, is constant, whatever that constant distribution is, then there is no change in heat over time; it requires changes in the spatial distribution in order to have changes over time. Our goal is to determine approximately at any time the distribution of heat in the bar, given initial conditions and boundary conditions.

We have three basic sets of decisions to make in modeling the distribution of heat in the bar: how

to represent the distribution of heat for computational purposes, how to define and model the thermal properties of the bar, and how to display our results in order to communicate the behavior of temperatures in the bar.

For the first decision, we could solve the differential equation directly, or we could model the heat transfer process by modeling the bar as a collection of cells and modeling the heat transferred from one cell to another adjacent cell as being proportional to the heat in the original cell. If two adjacent cells have the same heat, then the effect will be that they exchange the same amount of heat and end up with the same heat each started with.

For the second decision, we will choose to model the thermal behavior by a diffusion process. In the standard case of a cell that is adjacent to four other cells of similar properties, if the proportion of heat energy retained by a cell is α, then the proportion to be transferred to each of the adjoining cells is $(1-\alpha)/4$. We start with an initial condition and update all the cells' heat from that condition, replace the computed values with any values that are assumed to be constant (e.g. the cells at the positions of fixed-heat points), display the values in the cells at the end of this update, and take the resulting condition as the initial condition for the next round of the simulation. We may make the value of α a constant for all the cells, or we may set different values of α for different cells if we want to model a heterogeneous material. For an actual implementation of this problem, we must decide whether the bar itself is homogeneous or heterogeneous, where the connectors will be attached, and what the heat properties of the connectors will be. In the simplest case we would have a homogeneous bar with connectors at fixed points and fixed temperatures, and this case will be discussed below. We could also consider the case of heterogeneous materials and will suggest how a student might address the case of varying temperatures or connections from the connectors.

For the third, we need to go back to the discussion of visual communication, where this kind of problem was discussed. There we saw that we could represent the temperature at points in the bar either through color or through height, and we discussed how each of these can affect what the viewer sees in the image. In the current case, we will use both color and height to show how the temperature varies in the bar; this seems to be the strongest visual presentation of the information. The results are shown in Figure 7.2.

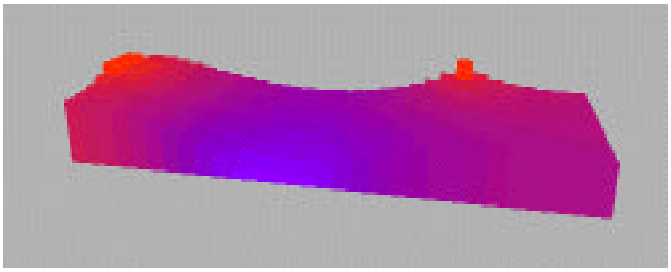


Figure 7.2: a simple representation of the temperatures in a bar with fixed-temperature connections

For the simplest case we saw above, the results are pretty simple to understand; the bar is hottest next to the points where the hot points are attached and is coldest next to the points where the colder points are attached. We do not show a legend that correlates the temperature with the color, and we should do that to make the figure more informative. We also do not show any change in the position or temperature of the attachment points, although there might be changes in the setup of this problem over time. These changes in temperature or position of the attachments could be controlled by user interaction to achieve various effects in the bar, and we invite the student to try some possibilities and see what happens. Specifically, we suggest that you look at a bar with the attached points at one end and with the constant temperatures at those attached points alternating

periodically between hot and cold. You should be able to see waves of heat and cold moving down the bar from the attachment points, and you should then try to explain why you see the behavior the display shows.

For the somewhat more complex case of a heterogeneous material we suggested, suppose that we are concerned that some region of the bar not get too hot, at least not very quickly, no matter how much heat is input into it at some fixed hot spots. We know that we can create a composite bar that includes segments with different thermal qualities. Let us assume that the node connections are on one end and let us define a bar that contains an insulating section between the two ends. In this case we will have much less heat transmitted by the insulating material than we would in a bar that had no insulator, and we would see a much slower heating of the material on the side away from the connections. We can model this kind of heterogeneous material more generally and store information on the heat properties of the material in each cell of the bar, adjusting the way we model the spread of heat and getting a good understanding of the way heat moves in such an object.

These examples show the qualitative behavior of the temperature distributions in the bar fairly well, but how accurate are they, and how could we have made them more accurate? We have used a discrete approximation to a continuous process, and we have approximated the physical nature of the heat transfer with a diffusion model, so we know that we do not have an exact solution to the problem. We can compare the results of our model with the results of experimental measurements, however, to allow us to make better approximations of the diffusion values, and we can increase the number of mesh points to reduce the error from the discrete approximation. This will usually allow us to draw good inferences from the model that we can use in understanding the nature of the heat distribution, and in many cases could allow us to design satisfactory materials to use in heat-sensitive environments.

Spread of disease

As another application of a diffusion approach, let's consider the behavior of a communicable disease in a region made up of a number of different communities. We need not try to be specific about the illness, because there are many illnesses that have this kind of behavior, nor will we try to be too specific about the behavior of the communities being modeled. However, we will find that we need to make a number of assumptions to model this in a way that allows us to create simple displays.

The general assumption on which we will build the disease spread model is that the basic mechanism of disease spread is for an infected and a susceptible (uninfected and not immune) person to come in contact with each other. When that happens, we will assume that there is a certain probability that the uninfected person will become infected. Because we cannot model every possible meeting of individuals, we will assume that the number of meetings between two populations is proportional to the product of persons in the populations. We will further assume that the region is divided into a grid (2D array) of rectangles with one community per rectangle, and that meetings only happen between members of adjoining communities, where adjoining is assumed to mean communities in the same row or column, differing in position by only one index.

As for the disease itself, we will begin by assuming that the initial population in each community is assumed to be entirely susceptible to the disease. We will further assume that the disease is not fatal, and once an individual has recovered from the disease (s)he is permanently immune to it. We assume that an infected person recovers with a probability of β (this value also models the duration of the illness), and that a susceptible person becomes ill with a probability of α times the number of possible meetings between that person and an infected person. We will assume that the number of possible meetings is defined as above.

With these assumptions, we derive a model that contains an array of susceptible, infected, and immune persons that represent the number of persons in each community in the 2D grid. For each time step in the simulation, we are able to calculate the number of meetings between infected persons and susceptible persons for each grid point by using the product of susceptible persons at that point and the numbers of infected persons at each neighboring point (including the point itself). We update the number of immune persons by calculating the number recovering and subtracting that number from the infected group, and then we add in the number of new infections from the group itself and from each of the neighboring groups. Finally, we subtract the number of newly-infected persons from the number of susceptible persons. Once we have the new numbers for the new time step, the display is updated and re-drawn; Figure 7.3 shows a frame from the animation produced for the simulation. The simulation introduces one additional feature by including a region for which there is no contact (we color it blue to simulate a lake); this changes the shape of the spread but does not change the fact that the infection spreads to the entire region.

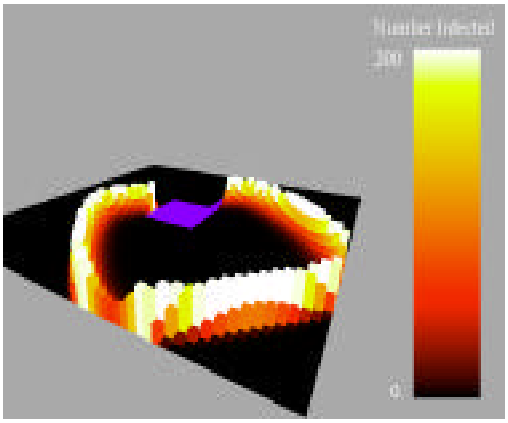


Figure 7.3: the disease spread model showing the behavior around the barrier region in blue

Note that the model predicts that everyone in each community will eventually become immune to the disease, and thus the disease will spontaneously disappear from the region permanently after the disease has run its course. This is evidently not a reasonable conclusion, but if we examine the weaknesses in the model we can see how its inaccuracies arise. We assume no births, because a newborn person has not yet acquired an immunity to the disease, and we assume no travel except between neighboring communities, so a new case of the disease would have to arise spontaneously somewhere in the region. So this model is overly simplistic, and yet it helps present the nature of communicable diseases at a simple level that can be refined by making a few additions to the model.

Function graphing and applications

We are certainly used to seeing many functions graphed with curves and surfaces in textbooks, so it would not be surprising to find that they are effective tools in understanding problems and finding solutions. In fact, we are so used to seeing problems phrased in terms of their graphs, curves, or surfaces that we may find it difficult to see a problem where these arise. But we will try.

Drawing a graph of a real function of a single variable is very simple. The graph of the function f is defined to be the set of points $(x, f(x))$ for values of x in the domain of the function. The graph is drawn by choosing sample values in the domain, calculating the function for each value, and drawing a straight line from one pair to the next. We are familiar with this kind of graphing from

elementary mathematics and will not describe it further.

Drawing a graph of a real function of two variables is somewhat more complex, but if the function is reasonably well-behaved, this is straightforward with the aid of computer graphics, The general idea of a graph of a function of two variables presented as a surface is straightforward. For each pair (x,y) in the domain we calculate a value $z = f(x,y)$, and the set of triples $(x, y, f(x,y))$ is the graph of the function. As we saw above, however, we do not draw every point in the graph; we choose regularly-spaced values in the domain and calculate the function for each value, and join the vertices defined by these points, taken four at a time over the grid in the domain, by two triangles in the surface. Each triangle is planar and can be displayed with your choice of solid color or color determined by a lighting and shading model, depending on the results you want. The end result is described by Figure 7.4 below in a simple case. Note that as we said, the basis for the surface is the set of rectangles in the domain, with each rectangle defining two triangles in the actual surface.

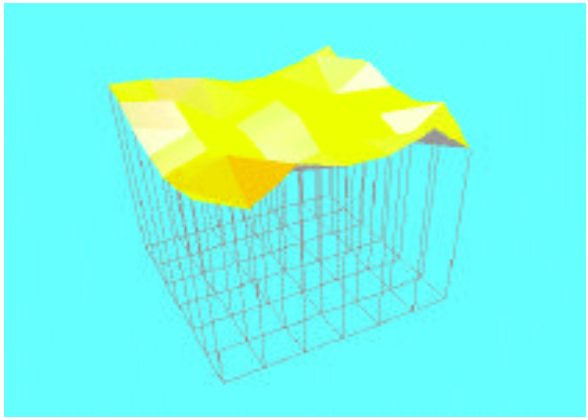


Figure 7.4: mapping a domain rectangle to a surface rectangle

An example of a straightforward surface graphed in this manner is the "ripple" function that shows rings proceeding outward from a central point, shown in Figure 7.5. We call it "ripple" because it looks like the ripples when a stone is thrown in the water. This is calculated as described above with the function graphed being $z = \cos\left(x^2 + y^2 + t\right)$. Over a square domain, $-5 \le x \le 5$ and $-5 \le y \le 5$, a double loop iterates both x and y and calculates the value of z for each pair (x,y). Once all the values are calculated and stored in a 2D array of z-values, we can iterate over the

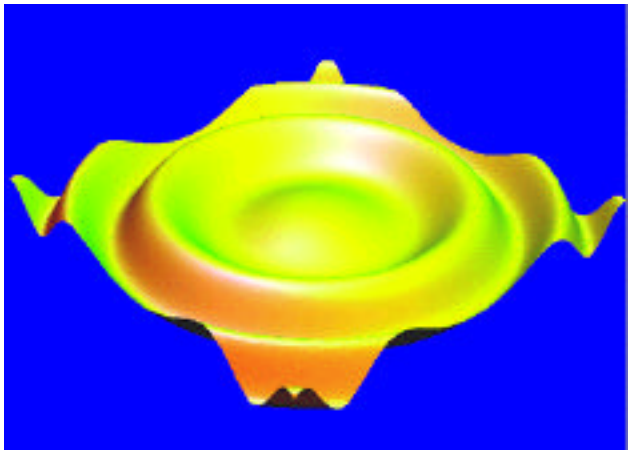


Figure 7.5: an example of a function surface display

domain again and draw the triangles needed for the surface. Note that this function contains a parameter t (think of time), and as this parameter is increased, the value of the argument to the cosine function is also increased. With this parameter increased linearly by adding a constant amount each time the image is re-drawn, the figure is animated and the waves move continually away from the center. The image here is drawn with a large number of triangles and using surface smoothing, so it is difficult to see all the individual pieces, but they are present if you look very carefully.

Many problems can be understood in terms of a mathematical function, and this function can often be understood better with the help of its graph. For example, let's consider the situation where there are various charges on a plane, and we want to be able to place these charges to achieve a certain electrostatic potential at a given point. We would start by realizing that the scalar electrostatic potential at a point (x, y) with charges Q_i at points (x_i, y_i) is given by Coulomb's law:

$$P(x,y) = \sum \frac{Q_i}{\sqrt{(x - x_i)^2 + (y - y_i)^2}}$$

For any set of fixed charges at fixed points, this defines a function of two variables that can be graphed as noted above. This function is fairly simple, but it's not at all clear from just the equation what the nature of the electrostatic potential is. For a particular configuration with one positive charge and two negative charges at given points, the graph in Figure 7.6 describes the electrostatic potential in a rectangle, as described earlier in the chapter on visual communication. From this graph we see clearly that the potential looks like an elastic sheet with spikes directed upward or downward depending on the sign of the charge. The actual values of the potential at a point could be estimated from the 2D pseudocolor plane if we were to include a scale in the image.

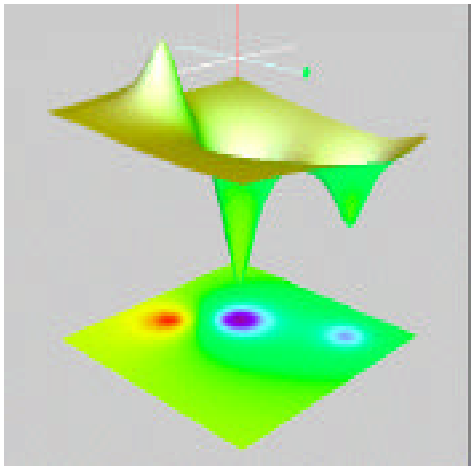


Figure 7.6: the coulombic surface from three point charges (one positive, two negative) in a plane, with both a 3D surface and a planar pseudocolor presentation

If you want to achieve a given electrostatic potential at a given point, you can start with this kind of graph and define a way to select one of the point charges to manipulate it. You can then move the point around, with the mouse, for example, or you can change the amount of charge at that point. Both of these will change the graph, allowing you to see how the potential at your particular point changes. With a little experimentation, then, you will be able to get the appropriate potential at your point of interest, and you can even experiment more to see whether there are other ways that could achieve the same potential in easier ways. Thus the image provides an interactive tool to create the potentials you want at the places you want them.

Another example that can show different behaviors via functions is that of interacting waves. Here we have two (or possibly more, though we will restrict ourselves to two for this discussion) wave functions, and the overall displacement they cause is the result of a sum of the two functions.

There are two kinds of waves we may want to consider: wave trains and waves from a given point. A wave train is described by an equation such as $f(x,y) = a\sin(bx + cy + d)$ that contains an amplitude *a*, a frequency and direction determined by *b* and *c*, and a displacement given by *d*. By incrementing the displacement with each successive re-drawing of the image, we can animate the wave behavior. The behavior of two wave trains is thus given by a function that is the sum of two such equations. In the left-hand side of Figure 7.7 we see the effect of one wave train of relatively high frequency meeting a wave train of about the same amplitude but a lower frequency at an angle of about 120°. It is common to model water waves as a sum of many different wave trains of different angles, frequencies, and amplitudes.

In another example, we consider waves from a given point, which behave much as the example shown in Figure 7.6 above. Here each wave is given by an equation whose general form is extended from the "ripple" example earlier as $z = a\cos\left(b((x - x_0)^2 + (y - y_0)^2)\right)$. Thus each wave function is defined by an initial point (x_0,y_0), an amplitude *a*, and a frequency *b*. Thus two waves with their own initial points, amplitudes, and frequencies are given by a sum of two of these functions. When two (or more) of these wave functions are added together, they create a complex pattern of interference and reinforcement, as indicated in the right-hand side of Figure 7.7 where the two waves have slightly offset centers and the same amplitudes and frequencies.

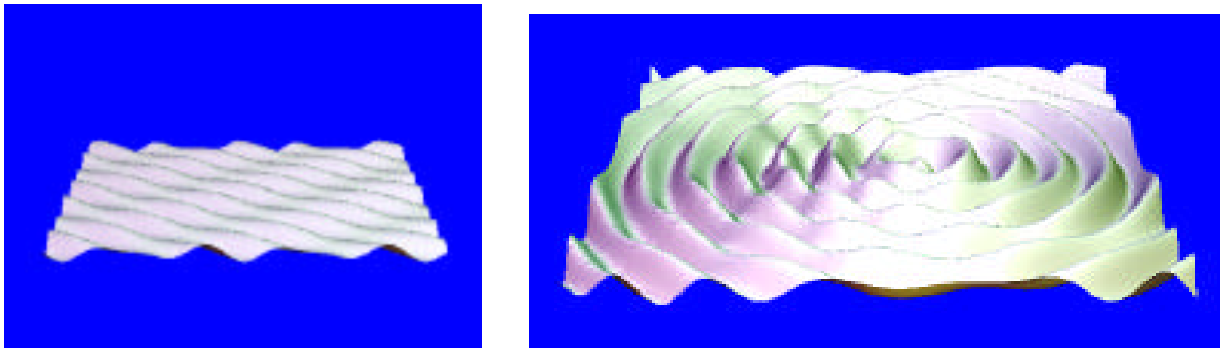


Figure 7.7: two wave trains intersecting at a shallow angle, left, and two circular waves whose origins are offset by 3π/2, right

Parametric curves and surfaces

A parametric curve is given by a function from a line or line segment into 2D or 3D space. This function might be analytic (given by a formula or set of formulas) or it might be an interpolation of data values. The former is probably easier to work with for now, but an example of the latter is found in a later chapter on interpolation and splines.

To consider an example of an analytic curve, let's consider what we would have in cylindrical coordinates if we increased one variable, the angle, at a slow rate while drawing circles with the other two variables. That is, if we name our parameter t and used the parametric equations

```
x = (a*sin(c*t)+b)*cos(t)
y = (a*sin(c*t)+b)*sin(t)
z = a*cos(c*t)
```

for some real constants *a*, *b*, and *c*. For the example shown in Figure 7.8, *a*=2.0, *b*=3.0, *c*=18.0. Then the parametric spiral shown moves around a torus as *t* takes on the values between 0 and 2π.

The parametric equations are evaluated at very small steps in the parameter *t*, giving a sequence of points that are joined by straight lines, yielding the rather smooth display of the curve. This spiral goes around the torus 18 times (c=18) while describing a circle with radius 2 and center three units from the origin, as you can see fairly readily from the figure.

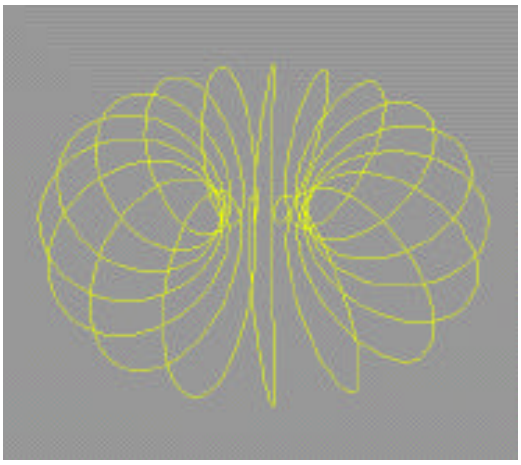


Figure 7.8: The toroidal spiral curve

Parametric surfaces can be a little more challenging. Here we take a 2D region in the plane and map it into 3D space in various ways. This kind of modeling may need some work to lay out, but you can achieve some very interesting results. In Figure 7.9 we show an example of such a surface; here we took a planar rectangle and divided it into three parts, folding the cross-section into an equilateral triangle. Then we twisted that triangle around 4/3 times and stretched it around a torus, putting the two ends of the triangular tube back together. The resulting surface is one-sided (you can trace all the surface without crossing any of the triangle edges) and is interesting to hold and manipulate; in the later chapter on hardcopy you can see photographs of the surface that have been created with various 3D hardcopy technologies.

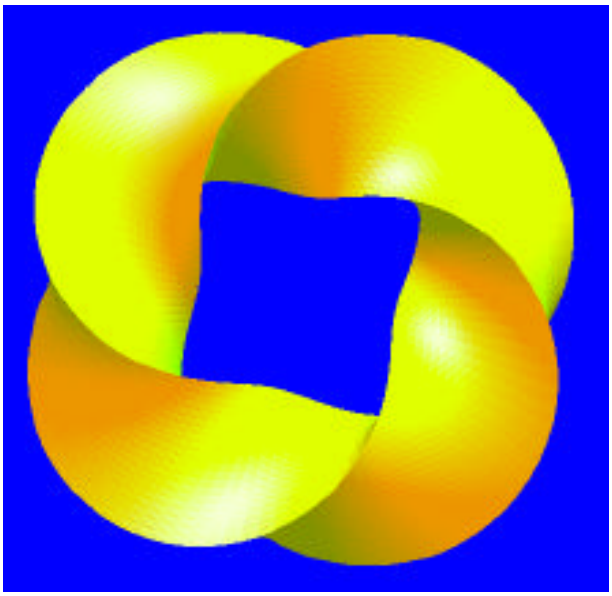


Figure 7.9: a 3D parametric surface of two variables

To look at this surface in more detail, the domain is the rectangle in parameters u and v defined for $-2\pi \le u \le 2\pi$ and $-2\pi \le v \le 2\pi$, and given by the equations

```
X(u,v) = (4+2cos(4u/3+v))cos(u)
Y(u,v) = (4+2cos(4u/3+v))sin(u)
Z(u,v) = 2*sin(4u/3+v)
```

This should look fairly familiar, because it is much the same as the toroidal spiral curve above. The difference, however, is that for the spiral we stepped the single parameter in small steps, while here we step the parameter *v* in small steps (e.g. 100 steps around the torus) while stepping the parameter u only three times, giving us large spaces between steps and making the cross-section of the surface triangular. This is shown in the layout of the parameter space shown in Figure 7.10; in general, it can be very helpful to lay out the parameter space in this way before going on to define the surface in more detail. In particular, you can see from the parameter space layout that the *u*-space is used to create the triangles and the *v*-space to create the steps around the torus. You might conjecture what could be produced if you used four steps instead of three in *u*-space (did you see a square cross-section?) or another number, but if you should do this, you must be careful to change the constant 4/3 in the equations so that you will still get a closed surface.

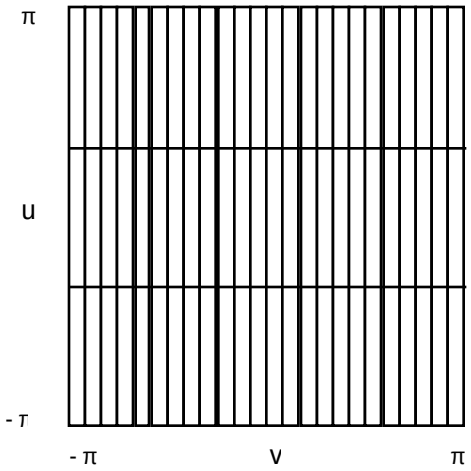


Figure 7.10: the parameter space for the surface shown in Figure 7.9, simplified by showing fewer steps in the v parameter

If you find that these parametric surfaces are fairly simple, you can test your geometric intuition by mapping your 2D domain into 4D space and seeing what you can produce there. Of course, you probably cannot display the 4D surface directly, so you will want to use various projections of the result into 3D space. There are some classical surfaces of this kind, such as the Klein bottle. A set of parametric equations (in 3-space only) for the Klein bottle are given by

```
bx = 6*cos(u)*(1 + sin(u));
by = 16*sin(u);
rad = 4*(1 - cos(u)/2);
if (Pi < u <= 2*Pi) X = bx + rad*cos(v + Pi);
else X = bx + rad*cos(u)*cos(v);
if (Pi < u <= 2*Pi) Y = by
else Y = by + rad*sin(u)*cos(v);
Z = rad*sin(v);
```

as translated from a fairly well-known Mathematica™ function. The left-hand image in Figure 7.11 below was obtained by replacing the functions for the torus in Figure 7.9 above with the functions given here, and changing the domain to $[0,2\pi]$ instead of $[-\pi,\pi]$. In fact, once you have a good program for parametric surfaces, it should be easy to adapt to different kinds of surfaces.

The actual construction for the Klein bottle is a little more complicated than shown above. The domain for the Klein bottle is a rectangle in 2D space, similar to the domain for the twisted torus above, but the function has one very different treatment for the domain: as shown in the right-hand image of Figure 7.11, with the sides identified as shown, the two sides labeled *b* are matched as they were when we created the cylinder above, but the two sides labeled *a* are matched in reverse order. This cannot be done in 3D space, but it can in 4D space, and the result has properties much like a cylinder, but with an embedding that can only be suggested by any 3D projection. The left-hand image of Figure 7.11 shows a standard 3D projection of the Klein bottle, but there are many others that are of interest and that illustrate other properties of the surface; for example, the Klein bottle may be created by gluing two 3D Möbius bands together in 4-space.

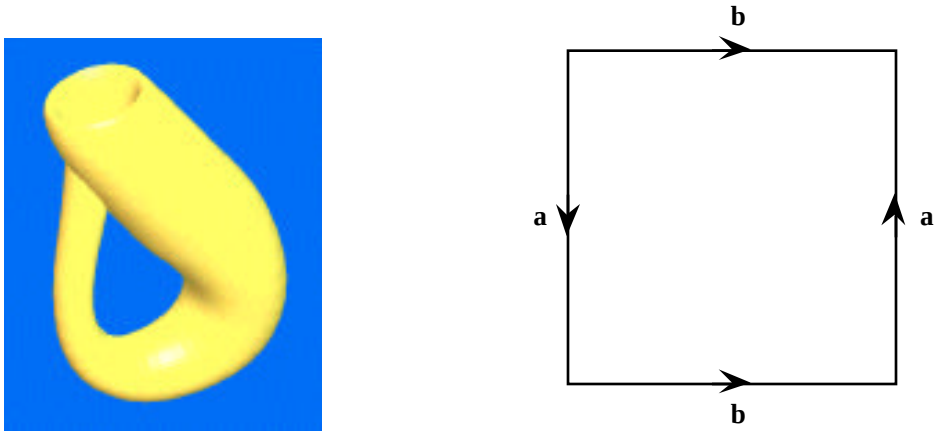


Figure 7.11: a 4D parametric surface, the Klein bottle (left) and the structure of the parametric region that defines it (right)

Graphical objects that are the results of limit processes

Sometimes curves and surfaces arise through processes that are different from the closed-form function graphing we described above. Some of these graphical objects are, in fact, quite surprising, such as some that arise from limit processes.

Two of these limit-process objects might be of particular interest. In the calculus, it is shown that while any function differentiable at a point must be continuous at that point, the converse is not true: you can have continuity without differentiability. There are some examples of continuous but nowhere differentiable functions given in most calculus texts, but it can be difficult to see the actual results of the process. One of these functions is called the blancmange function because its graph—the surface defined by the function—looks somewhat like the very lumpy blancmange pudding often served in England at holidays. This surface is defined recursively as the sum of an increasing number of piecewise bilinear functions that connect the points $(i/2^k, j/2^k, z)$ at level k, where z is 0 if i or j is even and $1/2^k$ if i and j are both odd. Because the sum at any point is not larger than $\sum_k 1/2^k$, which is a converging geometric sequence, this sum converges and the surface is well defined. However, within any neighborhood of any point (x,y) there will lie many points $(i/2^k, j/2^k)$ for even values of i and j, and at each of these points there is a discontinuity of one of the summands and hence of the function. Because there are discontinuities within any neighborhood of any point, the function cannot be continuous anywhere. The graph of this function is shown in Figure 7.12. For more on the blancmange function, see [Tall]; a very similar surface, called the Takagi fractal curve, is described in Iterated Function Systems terms as an example of midpoint displacement processes in [Pietgen]; see Chapter 16 for more details.

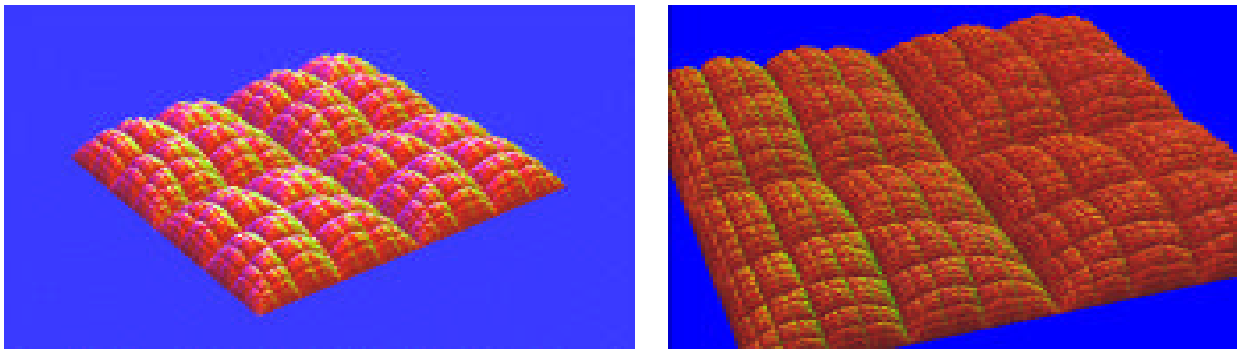


Figure 7.12: the blancmange surface, left, and zoomed in with more iterations, right

Another surprising object is called the Sierpinski gasket. It can be defined in many different ways but is always a limit of an iterative process. One definition is to take a tetrahedron and to replace it by four tetrahedra, each half the height of the original tetrahedron and occupying one of the corners of the original. This process repeats without limit, giving tetrahedra whose volume approach zero but which occupy positions along the edges of all the possible sub-tetrahedra in the space. Another definition which is somewhat easier to compute is to take the four vertices of a tetrahedron and any collection of points in 3D space. For each point, choose one vertex at random, and move the point one-half of the way towards the vertex. Do this many times, and the limiting positions of the points will lie on the Sierpinski gasket. For more details of this process, see [Pietgen] for the 2D case; the 3D case is a very simple extension. In Figure 7.13 we see the four vertex points of a tetrahedron in red, and points in cyan that have been calculated by starting with 50,000 random points and applying this process.

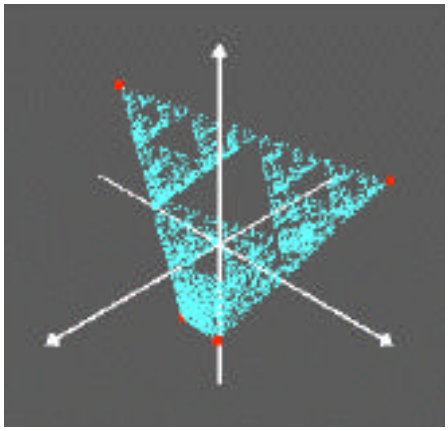


Figure 7.13: the Sierpinski attractor

Scalar fields

A *scalar field* is a real function of a variable on a domain, which is a very general principle. If the domain is a region in a 1D real space, it is called a 1D scalar field and defines an ordinary function of one variable. A 2D scalar field is a scalar field on a 2D domain, or a real function of two real variables, but this concept is much more general than the kind of surface we discussed above. Many kinds of processes can give rise to such a function, and those processes can produce functions that are far from the continuous surface we were working with. They can even give rise to functions that cannot be expressed in a closed form (that is, in terms of equations). 3D scalar fields are discussed below when we talk about volume data, because they are usually thought of in

volume terms.

An interesting example of a 2D scalar field in the digital elevation map (DEM). These maps, available from the USGS and other sources, are 2D grayscale images whose color value represents the elevation of points in a map space. They are, in fact, examples of images colored with pseudocolor data representing elevations. If you know the base and top of the elevation scale for a map, then, you can read the pixels and calculate elevations for each point, giving you a surface grid that is a good representation of the topography of the area. In addition you can often get an aerial photograph, radar scan, or other image of the same area. Figure 7.14 shows the image of a digital elevation map and of an aerial photograph of a portion of the same space, specifically of the San Diego State University campus area, which you will find toward the lower right of Figure 7.15.

Figure 7.14: height field from USGS digital elevation map (left) and texture map of a portion of the same space from aerial photographs, courtesy of Jeff Sale, San Diego State University

When the height data is turned into polygons and they are texture-mapped with the photographs, the results can be strikingly realistic. In Figure 7.15 we see an example of the San Diego area east of downtown, courtesy of Jordan Maynard; the San Diego State University campus can be seen near the lower right of the image. Note that because the image is texture mapped from a photograph, the actual buildings are not seen as they would be in an actual photograph; only the shapes of the buildings on the ground are shown.

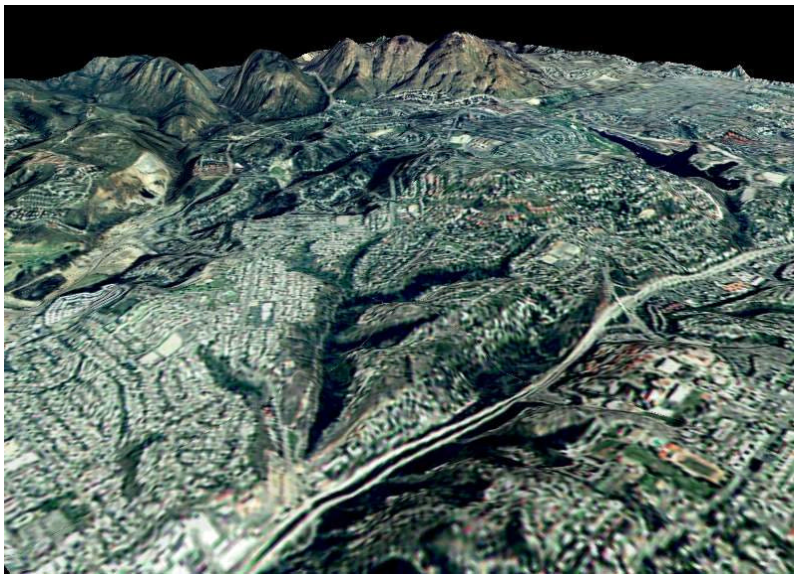


Figure 7.15 height field and texture map made into a terrain visualization

There are a number of other places where values are sampled on a domain, and we can think of these as samples of a scalar field. For example, laser range scanning produces a scalar field for distances from a sampling point; as in the terrain map case the sample points may be converted into geometric vertices of polygons of the space being scanned. As an example of this approach, the research into the paleolithic cave art of Cap Blanc used both photographs of the cave, as shown in Figure 7.16.

Figure 7.16: Photograph of cave site with photogrammetric reference card and laser scan point

The cave's geometry is computed from the results of a laser scan carried out in the cave. The scanner captures the vertical and horizontal offset from the original scanner position at each of many hundred systematically-scanned points. This produces a height field shown as both raw laser scans (left), calculated from the offsets and distance. This is further developed into a mesh (right), both shown in Figure 7.17.

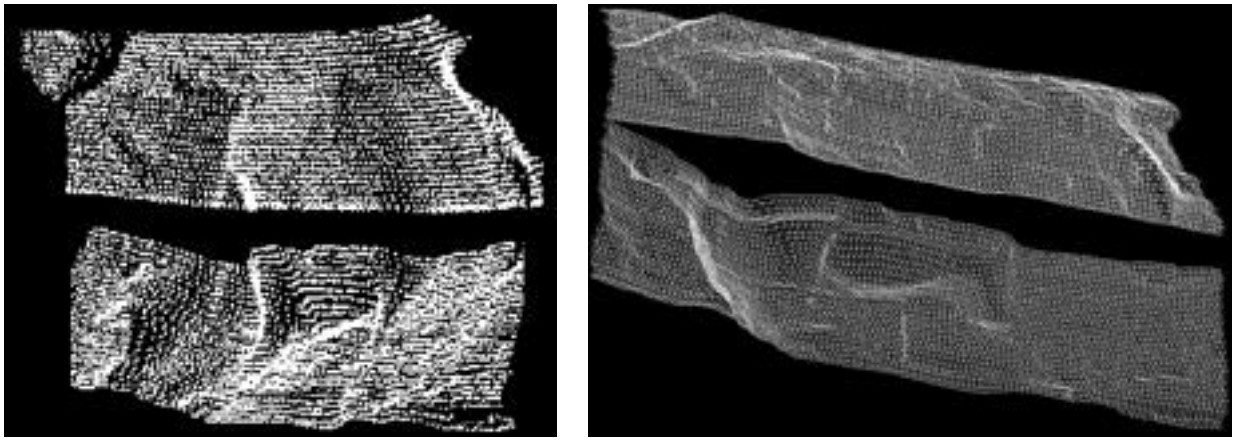


Figure 7.17: original laser scan data (left) and the geometric mesh computed from that data (right)

The positions of each point on the mesh is then matched with a position on the photograph that acts as a texture map for the cave wall, allowing the researcher to create an accurate geometric and color model of the cave wall.

This model is then used to understand how the cave might have looked under different lighting conditions, so the researchers can study the effect of different kinds of firelight in various positions. An example of this is shown in Figure 7.18, and you will see how the shape of a horse stands out from the light of a lamp. As this lamp is moved, the horse seems to move, showing that

the original cave inhabitants may have had the notion of a living horse from the carving.

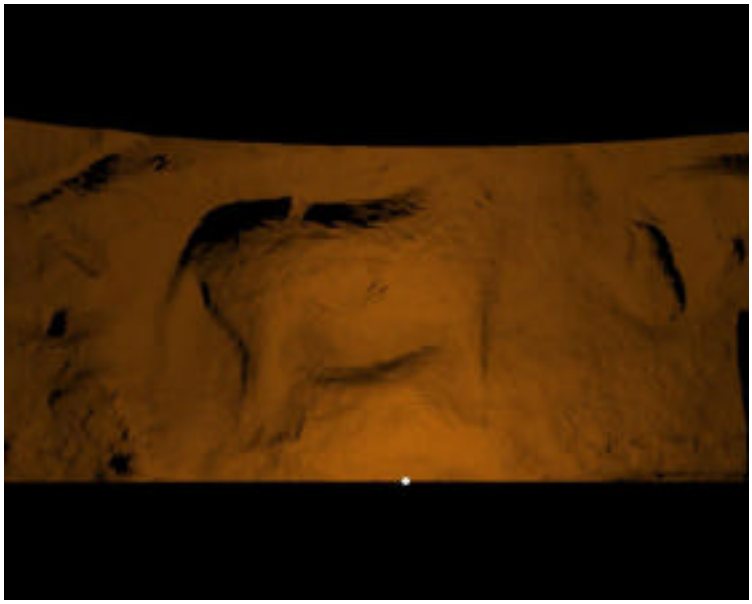


Figure 7.18: The cave wall shown with a simulated animal-oil lamp

Representation of objects and behaviors

In some cases it is possible to represent an object of interest and to represent its behavior so we can examine that behavior to understand a phenomenon. In these cases we can create a simulation of the behavior and our images help us to visualize the behavior as expressed in the simulation.

This is actually a very broad topic and we cannot go into much detail on simulations here. We will present two simulations of the behavior of an ideal gas and will show how we can not only visualize the behavior but also get data from the simulation to test hypotheses on the behavior. This allows us to get both numerical and visual results, allowing us to verify that the simulation has the right analytic properties while we look at its visual properties.

Gas laws and diffusion principles

The behavior of ideal gases under various conditions is a standard part of chemistry and physics studies. Our first simulation will put a number of objects (points, actually, but they represent molecules) into a closed space. Each will be given a random motion by generating a random direction (a vector with random coordinates, normalized to make it a direction) and moving the object a given distance in that direction. This simulates the behavior of a gas under constant temperature. We test for collisions of objects with the walls, and when these are detected the object direction is changed to simulate the object bouncing off the wall (however, we do not detect and account for internal collisions between objects). These are displayed as in Figure 7.19 and the number of objects hitting a wall is counted at each step. We allow the user to test the simulation by pressing a key and getting the pressure (the number of hits per unit area of the box), volume of the box, and the product. This product is PV, which should be a constant for an ideal gas.

This simulation would be of minimal interest if we left it as described, but we also allow the user to increase or decrease the volume of the box. If this is an accurate simulation of the behavior of an ideal gas, we should have the same product (within statistical limits) as we change the volume of the box, although we need to wait for the gas to expand or contract before we test the model. Thus

the display in Figure 7.19 shows the objects in the box (with one traced through several steps to show the random walk behavior of the gas) as well as the results of several tests.

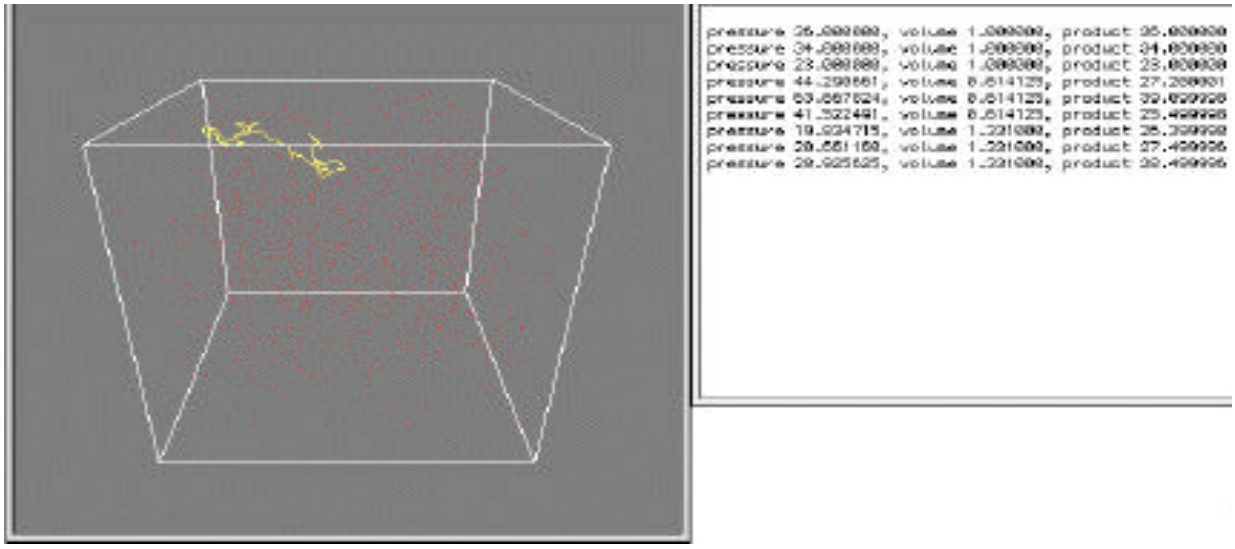


Figure 7.19: Displaying the gas as particles in the fixed space, with simulation printouts included

Another simulation we could create, building on the one above, examines the behavior of gas in a box that is divided by a semipermeable membrane. Such a simulation is shown in Figure 7.20. In such a membrane, if a gas molecule hits the membrane from one side, there is a different probability of it passing through the membrane than if it hits from the opposite side, concentrating that gas somewhat more on the side to which transmission is easier. If there are two different kinds of molecules in the gas and if the objects are treated differently by the membrane, we would expect to see some separation of the two kinds of molecules in the two parts of the space. We simulate this with particles simulating the gas molecules, as above, and using the same kind of code to detect particles hitting either a wall or the membrane. If a particle hits the membrane, a random number function determines whether the particle passes through the membrane or is reflected back. We are again able to tabulate the state of the particles in the system at any time, but in this case we count the number of particles on each side and calculate the ratio of particles on the left and right sides of the membrane. In the initial state, all the particles are on one side of the membrane; as the simulation runs, we expect to see a steady state reached with different left/right

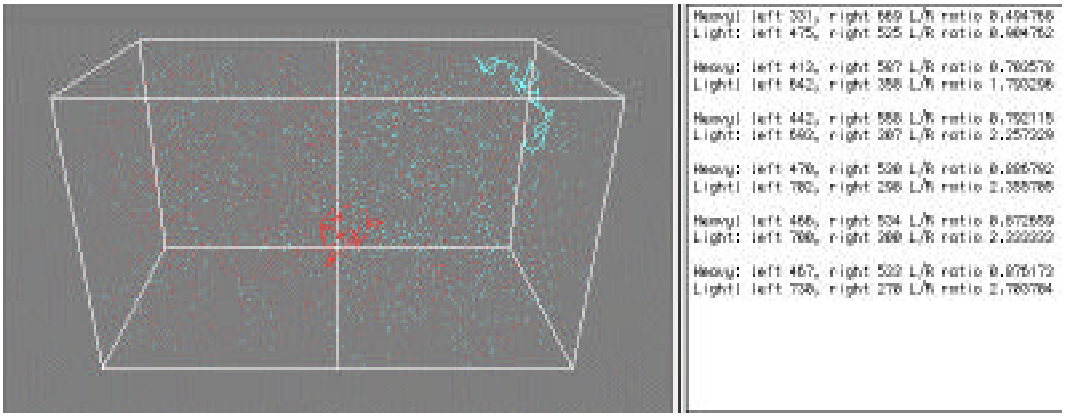


Figure 7.20: display of the diffusion simulation, directly across the membrane (including data output from the simulation)

ratios for the two kinds of gas. In the figure we see a snapshot of the space as one of the (red) particles has just passed through the membrane and returned, and we show the display of the tabulation at several times before the screen capture image.

Molecular display

Sometimes we will simply want to see some kind of invisible objects or processes in order to understand our problem better. This has long been the case in chemistry, where the ability to identify and display the structure of molecules has been an important part of chemistry students' education and where molecular visualization has led to great developments in drug design and other breakthroughs. Molecular visualization is a large topic, however, and many of the visualizations are very complex and require a deep understanding of molecular-level physics. We cannot hope to create this kind of complex work in a beginning computer graphics course, but we can show the beginning of the process.

The traditional start to understanding molecular structure (at least in the author's student days!) was the spring-and-ball display of a molecule. In this display, each atom is represented by a ball whose color represented a particular kind of atom, and each bond was represented by a spring that connected two balls. This allowed you to assemble a fairly simple molecule and manipulate it by moving it around and seeing it from various angles. We can do at least this much fairly readily, and perhaps we can suggest some ways this could be extended.

To begin, we need to know that the basic geometry of many different kinds of molecules has been determined and is readily available to the public. One of the major information sources is the *protein data bank* at `http://www.rcsb.org` (and at several mirror sites), and another is MDL Information Systems (`http://www.mdli.com`). Molecular descriptions are stored at these (and many more) sites in standard formats, and you can go to these sources, find and download descriptions of the molecules you want to examine, and create displays of these molecules. The descriptions are usually in one of two major formats: .pdb (protein data base) format or .mol (CT files) format. An appendix to these notes gives you the details of the formats, and your instructor can make available some very simple functions to read the basic geometry from them. Check out the "Molecule of the Month" listing from the University of Bristol for some interesting examples! (`http://www.bris.ac.uk/Depts/Chemistry/MOTM/motm.htm`)

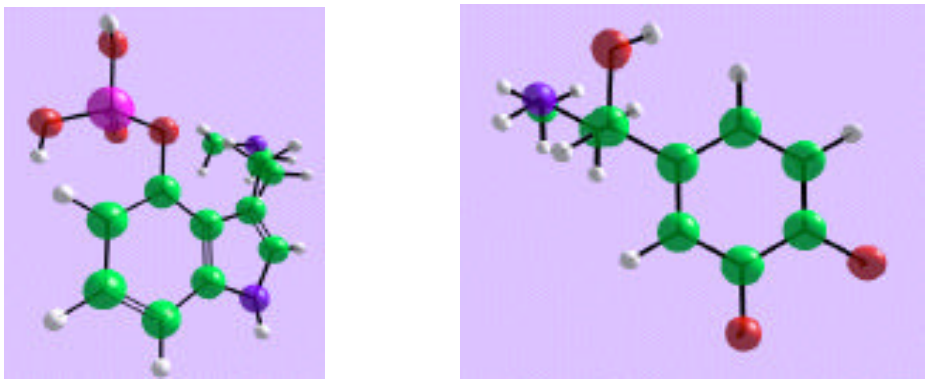


Figure 7.21: displays from psilocybin.mol (left) and adrenaline.pdb (right)

Creating a display from the molecule description is fairly straightforward. Once you have decoded the description file, you will have a list of atom positions and names, and a list of atomic bonds in the molecule. You can then draw the atoms at the positions indicated and draw in the links. It is common to draw the atoms as spheres with colors and sizes that are traditional for each kind of

atom; the spheres may be drawn as opaque or partially transparent, and the colors and sizes can be provided in the file reading function. Links are usually drawn as some kind of line. Figure 7.21 shows an couple of examples of simple molecules from the molecule reader provided with these notes and a display function written by the author. Note that the atoms are drawn with a fairly small alpha value so the bonds and other atoms can be seen; note also that in the example from the .mol file, double bonds are shown (these are included in the .mol file format but not the .pdb format). It is straightforward to include various kinds of interaction with these displays, as described in the section of these notes on creating interactive programs.

In more advanced work, it is common to use displays that include additional information, such as displaying the molecule as a smoothed surface around the spheres with colorings that illustrate the electrostatic forces at various places on the surface. This kind of display helps to show how molecular docking would happen by showing the surface shapes as well as the forces that would guide the docking process.

Monte Carlo modeling process

There are a number of interesting simulations that are built from generating large numbers of random occurrences and looking at the long-term behavior of the system. Because random numbers have long been associated with gambling, and because the casino at Monte Carlo is known worldwide, such processes are often called Monte Carlo processes.

These processes can range from the simple to the complex. We have already seen an example of such a process in the gas law and diffusion simulations above, but there are many other kinds of Monte Carlo simulations. In a very simple case, consider a 2D region that is bounded by a complex curve that makes it very difficult to measure the area of the region. If it is possible to determine quickly whether any given point is inside or outside the region, then a Monte Carlo approximation of the region's area could be made by generating a large number of random points in a known (probably rectangular) area containing the region. One then counts the proportion of the total number of points generated that lie within the region, and takes that proportion of the known area as an estimate of the region's area. A very similar process operating in 3D space allows you to estimate volumes, and is illustrated graphically by Figure 7.22 which shows 10,000 points in a cube that is two units on a side and contains a number of randomly-placed (and overlapping) spheres. The spheres are rendered in yellow with a low blending value, points lying inside the spheres are colored red, while those lying outside are colored green, so you can get a general idea of the relative proportions of the points. This figure is not oriented towards getting an exact solution, but it will provide a reasonable estimate that could be very helpful if you were trying to explain the concept and process to a layman (or, perhaps, a judge or jury). It is presented by a program that also allows the user to rotate the volume in arbitrary directions to see how the highlighted points lie in the volume.

More complex and interesting kinds of Monte Carlo simulations can be found in other areas. For example, in queuing theory and transportation engineering, arrivals are defined to be random with certain parameters (mean, standard deviation, probability distribution) and service or transit times are also defined randomly. The system being studied is driven by these events, and the nature of the system is then studied through its reaction to these simulations. These are probably beyond the scope of this discussion, but they provide interesting examples of the power of a simple technique and it can often be very enlightening to use computer graphics to display the ongoing state of the system during the simulation. For example, in a traffic simulation, you could use colors to display the traffic flow in the transportation system, with high-attention colors used to show problem areas and low-attention colors used to show smooth flow. The fact that these colors might be red and green, respectively, would be a nice link to the culture of traffic study! And, of course, statistical simulations can be made more and more complex, so that war games, business simulations, and large-scale economic simulations are all in some sense Monte Carlo models.

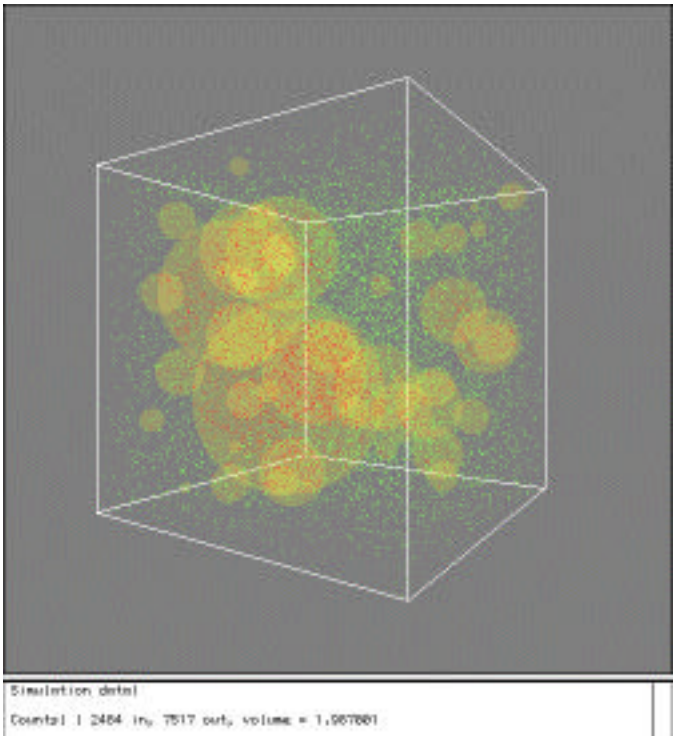


Figure 7.22 a Monte Carol estimate of a complex volume

4D graphing

Dimensions are an interesting question in computer graphics. We now say that graphics is natively 3D and we feel good about that because the world we live in seems 3D to us. Of course, it really isn't; it really has many more dimensions that we don't often think about. We think about driving as a 2D process because our streets (even in San Francisco) are laid out in a 2D pattern, but along with our position we also have velocity vectors, fuel levels, temperatures, and a number of other quantities that we must constantly balance while driving. When we start doing computer graphics to solve problems, we often find that our ability to define and view things in three dimensions is too limited to express everything we need. So as we discussed when we talked about visual communication, we also find that we must create models having more than three dimensions for many of our problems.

Volume data

Volume data is data that has one dimension, one real value, at each point in a volume. Extending the notion of a two-dimensional scalar field, this can be thought of as a scalar field on a three-dimensional space, and we will think of the data as coming from a scalar field, or a real-valued function of three variables. We will take two approaches to the display of the field: we will find implicit surfaces in the volume, or we will display the values in a cross-section of the field.

The implicit surfaces we will seek are surfaces made up of the points where the function has a constant value; these are also called *isosurfaces* in the volume. Finding them can be a hard problem, because volume data is unstructured and it is necessary to identify a structure in the data in order to create displays of these implicit surfaces. This is done in several ways, but the most common is the *marching cubes* process, where the volume is divided into a number of small cubes, called *voxels*, and each cube is analyzed to see whether the surface passes through the cube. If it

does, then a detailed analysis of the voxel is done to see for which edges the volume crosses the edge; this allows us to see what kind of structure the surface would have on that voxel, and the surface within the voxel is displayed. For a good reference on this process, see Watt & Watt.

The display is not too difficult for this beginning course, but the analysis is very detailed and its code expression would involve more coding than we want to have here. Instead of the marching cube process, we will simply identify those voxels that contain a point whose value is the one defining the implicit surface, and we will display those voxels in a way that implies the surface. The simplest such display is simply to place a small lighted sphere in the cube, as shown in the left-hand image of Figure 7.23. This gives us the general shape of the surface (though with very little detail) with a modest amount of shape from the way the spheres are lighted, and is a way to start the analysis of the scalar field in the cube. This can lead to an exploration of the space by using an interactive technique to change the value that defines the surface. By sweeping the value through a range that covers the volume being studied, we can see the overall shape of the scalar field and get a better understanding of the problem that generates the field.

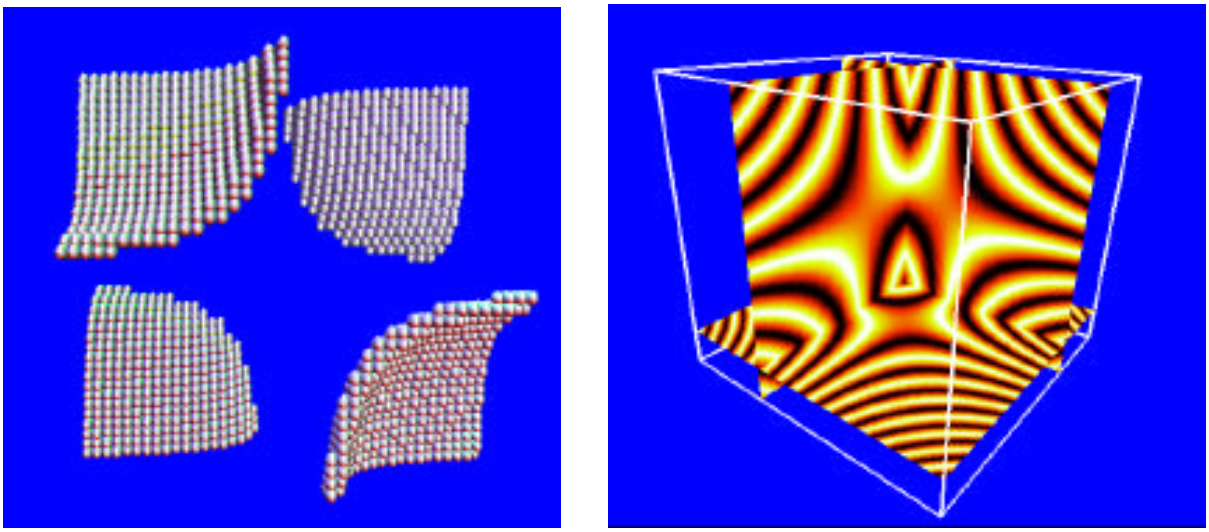


Figure 7.23: an implicit surface approximation that uses spheres to indicate surface location, left, and cross-sections of a function's values, right

Another way to understand the nature of the scalar cube is to slice the cube and see the scalar field on the slicing plane, or cross-section of the volume, as shown in the right-hand image of Figure 7.23. This allows us to think of the function as a 2D scalar field and to use whatever kind of mesh and pseudocolor we want on the slicing planes in order to see the field, so in some ways this is a more precise approach to the problem than the implicit surface. It also gives us an understanding of the relationships of the values throughout the space, in contrast to the shape of the level values, so it complements the implicit surface process well. Again, the display can be interactive so the user can explore the space at his or her leisure, and sweeping the planes through the space can give us an overall idea of the nature of the scalar field.

We should note that both the images in Figure 7.23 come from interactive programs that allow the user to sweep the space by increasing or decreasing the value defining the implicit surface, or that by moving move any of the three cutting planes parallel to the axes in order to see the entire shape of the scalar field. These interactive explorations are critical to understanding the data fully and are quite easy to implement. See the chapter on interaction for more details and examples.

The two images in this figure actually represent the same function: the three-variable hyperbolic

function $f(x,y,z) = xyz$. The implicit surface describes the geometry of the subspace where $f(x,y,z)$ is constant, which is a set of hyperbolas in four of the eight octants of 3D space. The four octants are those in which the signs of the variables are right for the sign of the constant, and the shape in each octant is determined by the constant. On the other hand, with the cross-sections we are looking for ways to represent the values in the 2D spaces that slice through a cube in 3D space through the use of a color ramp. We have used a rapidly repeating color ramp to show the contours of the scalar field, but a single color ramp across all the values in the cube would have been better if we had wanted to be able to read off actual values from the image.

Vector fields

We have discussed scalar fields earlier in this chapter, but we can extend the notion of function to include vector-valued functions on 2D or 3D space. These functions are called *vector fields*, and they arise in a number of situations. We should note that not all vector fields are necessarily presented as vectors; any function whose range lies in 2D or 3D space can have its values considered as functions, even if that might not be the intent when the function's context is set. For example, a complex function of a complex variable has a 2D domain and a 2D range because of the identification of complex numbers as pairs of real numbers. Let's consider some examples of functions that give rise to vector fields.

We already mentioned complex functions of a complex variable, so let's start there. If we consider the function $w = z^3 + 12z + 2$ described in [Braden], we see by analogy with real functions that as a cubic function, we should expect to have three "roots" of the equation, that is, three points where $w(z) = 0$. In the left-hand image of Figure 7.24, we take advantage of the alternate representation of complex numbers as $re^{i\theta}$ to display the graph of the equation by showing the magnitude of r as a color (in the uniform luminance color ramp defined in the chapter on visual communication) and by showing the direction θ as a direction vector. The three black points in the image correspond to the three roots we would expect, and we see that we have a smooth vector field across the space, showing that the complex function is smooth (indeed, infinitely differentiable) across its domain. With this display we can get a better understanding of the nature of the function and its behavior.

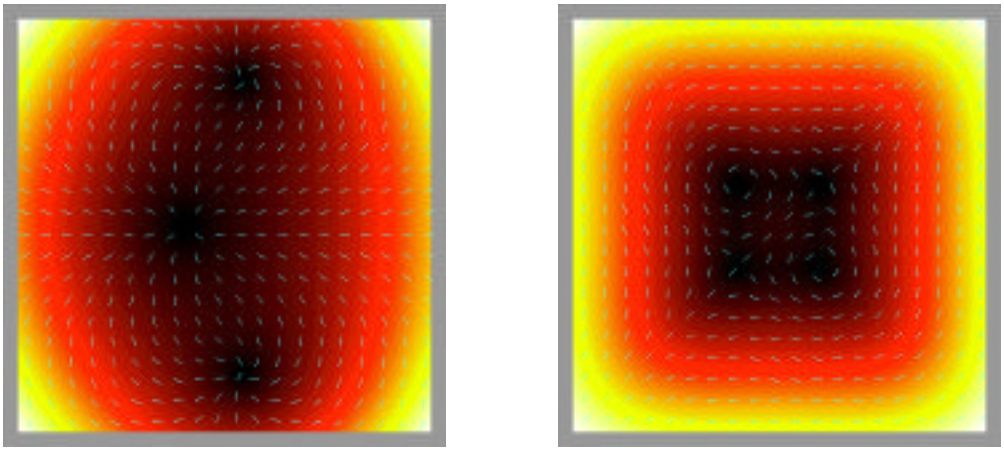


Figure 7.24: two visualizations: a complex-valued function of a complex variable (left) and a differential equation (right)

Another kind of problem generates differential equations that describe the nature of the change in a value across a domain. For example, if we think of fluid flowing across a plane, we see the velocity at each point of the plane; the plane is a 2D domain the velocity at each point is a 2D value.

The point is a position, and the fluid flow at that point can be seen as a 2D derivative of the position; to solve the differential equation would be to determine the function that describes the fluid flow. In the right-hand image of Figure 7.24, we see the pair of differential equations $x' = y^2 - 1$ and $y' = x^2 - 1$ [Buchanan] that describe such a situation, and the image shows the nature of the flow: low speed (magnitude of the velocity) at the four points (±1,±1), vortex flow at the two of these points where the signs differ, and source/sink flow at the two points whose signs are the same. As above, we use the color value to indicate the speed of the flow and the vector to show the direction.

Graphing in higher dimensions

We can go farther and consider functions on 3D space whose values are also in 3D space. This can challenge our imaginations because we are actually working in six dimensions, but such problems are everywhere around us. So standard examples include electromagnetic and electrostatic forces, gravitational fields, and fluid flow functions, especially when we consider fluid flow to include air as well as more usual fluids. Because we are considering examples of graphical thinking in such problems we must be careful about taking on examples that are too challenging compuationally, so we will not pursue the fluid flow problems; instead we will consider electromagnetic and electrostatic forces.

Consider electromagnetic fields generated by a current moving in a wire. These fields have vector values at every point in space, and so are represented by a function from 3D space to 3D space, and it takes six dimensions to represent this function fully. The function can be shown by sampling the domain on a regular grid and showing the field vector at each point, as shown in Figure 7.25 below. Such vector fields are commonly used, but they are difficult for a user to understand and there are various other techniques that can be used to show this information.

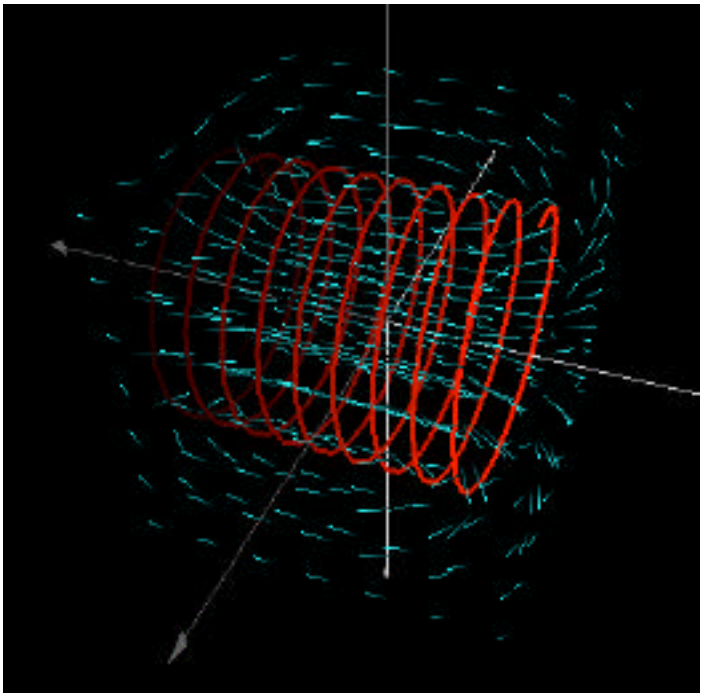


Figure 7.25: a magnetic field around a coil with a moving current

Another approach to a problem with 3D values in 3D space is to use a set of vectors to see the effects of the function. We saw the use of vectors in a 2D problem in 2D space earlier, but for the

3D problem let's return to Coulomb's Law of electrostatic force that we saw in scalar form earlier in this chapter. This law states that the electrostatic force between two particles is given by $F = kQq/r^2$, where q and Q are the charges on the two particles, r is the distance between the particles, and k is an appropriate constant. This force is directed along a vector from one particle to the other, and the value computed above may be taken as the length of that vector.

If we extend this to 3D, we start with a unit positive charge Q for one particle at an arbitrary point in space and calculate the force between that particle and a a set of particles with charges q_i in space. We get the equation: $F(x,y,z) = \sum_i kq_i V_i / r_i(x,y,z)^2$ for the force vector $F(x,y,z)$ at any point in 3D space, where V_i is the vector from the arbitrary point to the i^{th} particle and $r_i(x,y,z)$ is the distance between the point (x,y,z) and the i^{th} particle. In Figure 7.26, we show the way this force is distributed in 3D space with a collection of cyan vectors in that space; the figure also shows a set of particles with charges of +2 (green) or -1 (red), and the path of a particle of charge +1 that is released at zero velocity within the space.

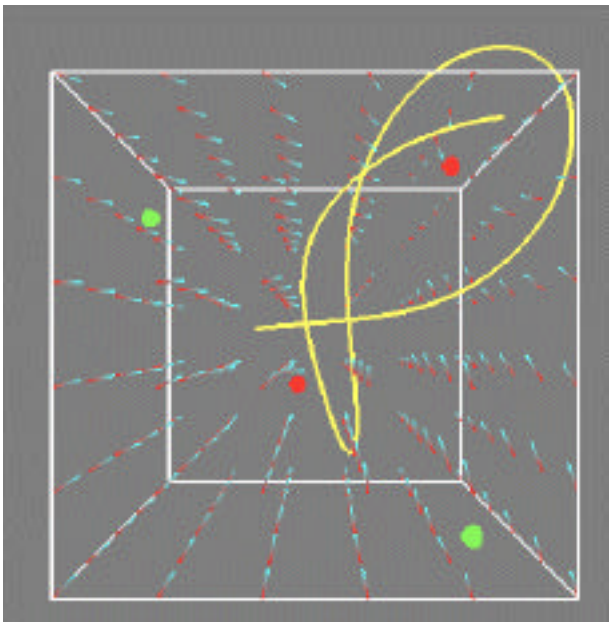


Figure 7.26: the 3D Coulomb's law simulation with a 3D vector field

There are many opportunities to choose different kinds of displays in these higher-dimensional cases. For example, just above we saw an example of displays of a 2D vector field on a 2D domain that separated magnitude and direction; we could examine a 3D vector field on a 3D domain by using 2D slices of the domain and on each slice displaying the 3D direction and the 1D magnitude of each vector. The opportunity to be creative here is very large.

An example of a field that has a long history of multi-dimensional data is statistics. Here it is rare for a data set to have as few as three variables, so there has been a lot of work with adding extra information to 3D scatterplots to try to see more information and with allowing user exploration of data by selecting which variables or combinations of variables to use. One of the key features of this kind of display is that the viewer needs to be able to move around the space to see the scatterplots from various angles in order to see any structure that might be present. The emphasis on interaction in these notes will help you find ways to let your viewer choose and manipulate the

data and space where viewing happens.

Data-driven graphics

examples are short here -- need to get some that are examples of generally accessible data and that involve genuine problems that could be of interest ...

"For data with a few dimensions, scatterplot is an excellent means for visualization. Patterns could be efficiently unveiled by simply drawing each data point as a geometric object in the space determined by one, two or three numeric variables of the data, while its size, shape, color and texture determined by other variables of the data."

Figure 7.27: an example of ...

As we noted when we talked about visual communication, it is important that you tell the truth with your images. It is very easy to create smooth surfaces that contain your data and that seem to say that the actual data varies smoothly, but sometimes the data actually contains jumps and other artifacts that simply aren't smooth, and making a smooth surface implies a continuity of values that is not true. Simplicity and truth are more important than attracting the eye with an incorrectly defined image.

Code examples

We did not want to slow down the general discussion of the kinds of modeling and visualizations one could create to support problem solving by discussing implementation and code questions, but we believe it will be useful to include some discussion of implementation and some sample code that was used to implement the models that led to the images in this chapter. In general, the discussion and the code will not focus on the graphics itself but on the modeling and the processing to support that modeling. When we want to point out something about the graphics being used, we may use generic API calls from which it should be relatively easy to infer the OpenGL that would be equivalent. However, sometimes we may need to use the exact OpenGL to make the point we want you to see in the discussion.

Diffusion

Diffusion processes work on a grid, so the first task is to define an appropriate grid on some 2D rectangle. This grid needs to be mirrored by an array of real numbers that will hold the values of the variable on the grid, so we could use declarations such as the following to implement these:

```
#define LENGTH 50
```

```
#define WIDTH 30
float grid[LENGTH][WIDTH]
```
Once these are defined and the grid is initialized, we can think about the code that defines how the value diffuses in the material. When we do, however, we realize that we will have to use the values in the grid to calculate new values in the grid, which would instantly lead to errors if we used the newly-calculated values instead of the original values. So we will need to use a mirror of the grid to hold the values we are creating, and this must also be defined
```
float mirror[LENGTH][WIDTH]
```

In general, the processing on the grids will look like calculating weighted sums. While the actual working of heat depends on the difference between the temperatures of each pair of adjacent cells, we will calculate the heat based on only the heat in the cells; the working of the processing for the two cells will take care of the difference. The general calculation takes a sum $\sum_{adjCells} heatAvail * weight * temp(cell)$ where the value of `heatAvail` is the heat available for the adjacent cell, which depends on the material, and `weight` depends on the diffusion model we are using. At the beginning of the chapter we suggested that the heat we do not keep is shared equally between the four nearest cells. We thus need to deal with the notion of the available heat for specific cells. We will use the notion that for any material, there is a proportion of the heat in a cell that is retained by that cell. We will also note that the weight in the formula is that proportion of the heat given up that is shared with neighboring cells, or .25 in the case we are discussing. Thus the equation of heat for any cell becomes the proportion retained by the cell plus the portions obtained from the adjacent cells. This value is calculated and stored in the mirror array, and at the end of the computation, the mirror array is copied back into the grid array. In terms of rough coding, this is given by

```
for i
   for j {
      mirror[i][j]  = prop*grid[i][j];
      mirror[i][j] += .25*(1-prop)*grid[i+1][j];
      mirror[i][j] += .25*(1-prop)*grid[i][j+1];
      mirror[i][j] += .25*(1-prop)*grid[i-1][j];
      mirror[i][j] += .25*(1-prop)*grid[i1][j-1];
   }
for i
   for j
      grid[i][j] = mirror[i][j]
```

An alternate, and more flexible, approach would be to define a 3x3 filter (or a different size if you wish), or array of non-negative values that sums to 1.0, and multiply the `grid[][]` cells around the `[i][j]` position by the values of that filter. This is a classical process for many kinds of computation and should be in your bag of tricks, if it isn't so already.

Of course, this does not take into account the behavior at boundary cells, where you will have to change your logic so that heat is not received from the out-of-bounds cells (and where less heat is given away, presumably.) This is left as a logic exercise for the reader. Finally, in case any of the grid points are being held at a fixed temperature (for example, if a hot source or cold sink is present), these points need to be given their fixed values to maintain the fixed situation there.

In case the material is not homogeneous, we must take into account the different coefficients of heat of two adjacent grid cells, and use an appropriate coefficient for the heat transfer. This needs to be symmetric—the value used by one cell with its neighbor must be the same, no matter which individual cell has the higher or lower heat coefficient—and we would suggest that the minimum of the two coefficients would be used. This change could be applied to both homogeneous and non-homogeneous materials without affecting the homogeneous case, of course, so we commend it to

the reader. This would, of course, require the addition of an array of these proportions of retained heat (which we can view as values between 0 and 1), so we need to have declared

```
float prop[LENGTH][WIDTH]
```

and instead of working with a fixed value of `prop` in the computations, we would use the minimum of the proportions of the original point and new point, so for the first of the adjacent points we would see

```
.25*(1-min(prop[i][j], prop[i+1][j])*grid[i+1][j];
```

When we have the processing done, we will need to display the results of the computation. To do this, we will use a generic cube as our basis and will scale, translate, and color the cube to represent the grid cell we are drawing. Assuming that we have a cube of unit side that is defined in the first octant and has one vertex at the origin, and assuming that our grid is in the first quadrant and starts at the origin, with each grid length being L, we have

```
for i
   for j
      set color to colorRamp(grid[i][j]);
      set translation(i*L, j*L);
      set scaling(L, L, L*grid[i][j]);
      draw cube;
```

If you have defined a reasonable view and set things such as depth buffering so it doesn't matter in which order you draw the geometry, this should provide a good image. You can then add the user interaction you might want to allow the viewer to examine the situation more accurately. You should also use the idle event to keep updating the image, because unless you started with a completely even temperature across the entire grid, it will take some time for temperatures to even out or to observe other temperature behaviors, depending on your simulation.

Function graphing

This graphing process was introduced briefly above. It is based on building a uniform grid in the domain space, somewhat like the grid used for the diffusion simulation above, and calculating the function values at each point in the grid, then using the grid and the function value to calculate quads and triangles that make up the surface. Here we will give a little more detailed description of the coding process in the following code sketch. The additional detail is included because this is such a fundamental operation that you must be absolutely sure you understand all the details. This sketch is written assuming that all the setup and declarations have been done, and using some meta-API statements to draw triangles and the like:

```
// assume a function calcValue(x,y) that calculates a function for each
// point in the domain; assume further that we are using the same number
// of points in each direction in the domain, and save the calculated
// values in a 2D real array values.  Assume functions calcXValue and
// calcYValue that compute the x- and y- values of the grid points in the
// domain.  Note that we use one more point in each of the directions than
// the number of rectangular regions we will create.
for ( i=0; i<=NPTS; i++ )
   for ( j=0; j<=NPTS; j++ ) {
      x = calcXValue(i); // calculate i-th point in x-direction
      y = calcYValue(j); // calculate j-th point in y-direction
      values[i][j] = calcValue(x,y);
   }
// with the values now calculated, create the surface mesh by creating
// two triangles for each rectangular piece of the grid.  We work in the
// counterclockwise direction based on looking down on the grid.
for ( i=0; i<NPTS; i++ )
```

```
   for ( j=0; j<NPTS; j++ ) {
      // calculate the x and y coordinates of the corners of the rectangle
      x0 = calcXValue(i);
      x1 = calcXValue(i+1);
      y0 = calcYValue(j);
      y1 = calcYValue(j+1);
      // draw first triangle
      beginTriangle();
         // calculate properties of the triangle such as its normal;
         // this is omitted here
         setPoint(x0,y0,values[i][j]);
         setPoint(x1,y0,values[i+1][j]);
         setPoint(x1,y1,values[i+1][j+1]);
      endTriangle();
      beginTriangle();
         // calculate properties of the triangle
         setPoint(x0,y0,values[i][j]);
         setPoint(x1,y1,values[i+1][j+1]);
         setPoint(x0,y1,values[i][j+1]);
      endTriangle();
   }
```

Parametric curves and surfaces

For parametric curves, it's straightforward to divide the domain, which will be an interval [a,b] on the real line, by defining a number of points along it at which to evaluate the parametric functions. If you use the equation $ta + (1-t)b$ for the interval, the domain consists of all point for values of t in [0,1]. If you want to divide the interval into N equal pieces (it's probably easiest to use equal subdivisions, but it's not required), then for values of i between 0 and N, the i^{th} point is simply $f(ta + (1-t)b)$ where we assume that the function f produces points instead of single values, and the curve is drawn by drawing lines between points determined by these values when they are sent to the function that defines the curve. Note that if the parametric curve lies in either 2D or 3D space, the process will be just the same. If you wanted to use unequal subdivisions along the domain interval, the computation of each point in the interval will be different, but drawing the curve by connecting the function values for each point will be the same.

Code to carry this out is quite straightforward, assuming that we have parametric functions $fx(t)$, $fy(t)$, and $fz(t)$ that determine the x, y, and z coordinates of a point on the curve:

```
#define START 0.0
#define END 1.0   // could be any values to start, end interval
beginLines();
   x = fx(START); y = fy(START); z = fz(START);
   setPoint(x,y);
   for i from 1 to N, inclusive
      t = START*(N-i)/N + END*i/N;
      x = fx(t); y = fy(t); z = fz(t);
      setPoint(x,y,z);
endLines();
```

For parametric surfaces the process is a little more complicated, but not much. We assume that the domain of the curve is a rectangle in u,v-space, with $a \le u \le b$ and $c \le v \le d$, and that we want equal spacing on each of the intervals for the surface. (The spacing may be different for each interval, of course; recall the discussion of the triangular cross-section for the torus.) We then

determine the points (u_i, v_j) in the rectangle that represent the i^{th} point in the u direction and the j^{th} point in the v direction, compute the surface coordinates from these points, and create quads (that will actually be drawn as pairs of triangles) from the four surface points corresponding to the four points that make up a small grid unit in the domain. These triangles may be drawn with any of the properties within the capability of your graphics API. The code for this operation is very much like the code for the triangles in the function surface above, except that all three coordinates of every point on the surface are determined by the function, not just the single coordinate.

Limit processes

Limit processes are something of a contradiction for computation, because computation is always finite while limit processes are almost always infinite. However, once we realize that converging limit processes will become arbitrarily close to their final state in a finite amount of time, it is clear that we can compute very good approximations to the final results and show these approximations to the audience.

For the idea of a limit curve or surface, then, we simply take the process out to as many steps as we wish, realizing that more steps can involve more time and perhaps memory. Once we have done the finite calculations, we simply have a function value (or perhaps several functions for a parametric operation) and we can use that in making our image as above.

Other kinds of limit processes, such as the Sierpinski attractor, will involve other operations that we cannot speak of generally, For the Sierpinski case, the process determines the positions for individual points and we simply draw them where they occur. The update for each step is done by an operation like the `idle()` callback discussed earlier and again in more detail in the chapter on event-driven programming. This callback simply needs to include the operations to change each point's position, and then call a redisplay operation to put it on the screen. As a code sketch, we have the following, where `vertices` is the array of vertices of the tetrahedron:

```
float points[3][N], vertices[3][4];
// in the display function, we find
beginPoints();
   for i = 0 to N
      setPoint(points[0][i], points[1][i], points[2][i]);
endPoints();
// in the idle() function, we find
for i = 0 to N {
   j = (int)random()%4;
   for k = 0 to 2 {
      points[0][k] = (points[0][k] + vertices[0][j])/2.0;
      points[1][k] = (points[1][k] + vertices[1][j])/2.0;
      points[2][k] = (points[2][k] + vertices[2][j])/2.0;
   }
}
post a redisplay event
```

Scalar fields

A 1D or 2D scalar field is essentially the same as a function graph or surface except possibly that there is another way to determine the scalar value at each point of the domain. Thus displaying a 1D or 2D scalar field is covered by our discussions above. 3D scalar fields are covered under 4D graphing below.

Representation of objects and behaviors

This section is really about displaying the behavior of simulations and the objects that they use. The examples we give here are fairly simple graphically, and our main challenge is to handle the details of the display. We choose to display a relatively small number of points in the volume because we don't want to clutter it up and lose track of the fact that individual particles are being tracked in the simulation. We generate the random motion of the particles by adding a small random number to each of the coordinates of each particle in the system, and we pick individual particles to track through time by maintaining a trail of their positions.

Probably the most interesting feature of these two simulations is noting and responding when a particle hits the boundary of a region. In the gas law example we have the particle bounce back into the volume; in the semipermeable membrane example we do the same for the walls, but for the membrane boundary we generate a random number and let that determine whether the particle penetrates or bounces off the membrane. In this sense the simulation has some Monte Carlo aspects, and we describe those below.

We are able to detect a situation when a particle would leave the volume under its normal random motion. In that case, we register a hit by incrementing an accumulator, which is simple; we calculate the point to which the particle would bounce by generating a pure reflection, assuming that particles obey the usual "bounce" rules. The reflection is straightforward, but you probably want to look at the geometry of a bounce situation a bit before you read the brief code below, which assumes that the region is bounded by walls a constant distance from the origin and that the array `p[][]` contains each of the coordinates of each point, in turn:

```
typedef GLfloat point3[3];
point3 p[NPTS];
if (p[i][j] > bound)
   {p[i][j] = 2.0*bound - p[i][j]; bounce++;}
if (p[i][j] <-bound)
   {p[i][j] = 2.0*(-bound) - p[i][j]; bounce++;}
```

Drawing the trails for individual points is straightforward; we simply maintain an array of the last N positions the particles have had and every time we generate a new display, we move each position back one in the array, put the new position at the front, and draw a set of connected line segments between points in the array. This is very helpful in showing the behavior of an individual point, and helps make the simulation display much more understandable to the viewer.

Finally, we gather the various statistics (how many particles are where, how many hit the walls of the volume, etc.) and display them either with the graphics system, as is described in the previous chapter on visual communication, or by printing them to the text console. This is triggered by an event of the programmer's choice, though we usually use a simple keystroke event.

Molecular display

Our displays of molecules are driven from arrays created by the functions that read the .pdb and .mol file formats, as noted in the earlier discussion. These arrays are of the form

```
typedef struct atomdata {
                        float x, y, z;
                        char  name[5];
                        int   colindex;
                        } atomdata;
atomdata atoms[AMAX];
typedef struct bonddata{
                        int first, second, bondtype;
```

```
                        } bonddata;
bonddata bonds[BMAX];
```

Here the field `colindex` in the atom structure will be found by looking up the name in the lookup tables described below. This index will then be used to find the color and size that matches the appropriate atom for the display.

The functions read the files and store the results in arrays of these structures, as indicated in the declarations. At the next step, the arrays are traversed and additional information is gotten from lookup tables that hold information such as size and color for individual atoms. The first stage is to look up the atom by its name and return the index of the atom in the tables. After this information is stored in the array, the images are created by traversing the arrays and drawing the molecules with size and color from the tables with this index; a partial sample of these lookup tables is below, with the first table being used to match the name, and the others used to get the color and size associated with the atoms.

```
char atomNames[ATSIZE][4] = { // lookup name to get index
   {"H   "}, // Hydrogen
   {"He  "}, // Helium
   {"Li  "}, // Lithium
   {"Be  "}, // Beryllium
   {"B   "}, // Boron
   {"C   "}, // Carbon
   {"N   "}, // Nitrogen
   {"O   "}, // Oxygen
   ...
   };
float atomColors[ATSIZE][4] = { // colors are arbitrary
   {1.0, 1.0, 1.0, 0.8}, // Hydrogen
   {1.0, 1.0, 1.0, 0.8}, // Helium
   {1.0, 1.0, 1.0, 0.8}, // Lithium
   {1.0, 1.0, 1.0, 0.8}, // Beryllium
   {1.0, 1.0, 1.0, 0.8}, // Boron
   {0.0, 1.0, 0.0, 0.8}, // Carbon
   {0.0, 0.0, 1.0, 0.8}, // Nitrogen
   {1.0, 0.0, 0.0, 0.8}, // Oxygen
   ...
   };
float atomSizes[ATSIZE] = { // sizes are in angstroms
   {0.37},         // Hydrogen
   {0.50},         // Helium
   {1.52},         // Lithium
   {1.11},         // Beryllium
   {0.88},         // Boron
   {0.77},         // Carbon
   {0.70},         // Nitrogen
   {0.66},         // Oxygen
   ...
   };
```

From this, it is pretty straightforward to draw the atoms, and the bonds are simply drawn as wide lines between the locations of the atoms whose index is defined for each bond. In case a double bond is indicated, two lines are drawn, each slightly offset from the atom center. Because the viewer will doubtless want to examine the structure of the molecule from all viewpoints, it is useful to allow arbitrary rotations. It can also be useful to allow alternate descriptions of the atoms (more or less transparency, larger size to get the space-filling kind of representation, etc.) by providing a

user selection through a control panel or a menu.

Monte Carlo modeling

We are using the term Monte Carlo rather loosely to refer to any kind of process based on random values. In that sense, the gas law and semipermeable membrane simulations were Monte Carlo models, and were so noted in their discussion. Sometimes, however, Monte Carlo simulations are taken to mean simulations where events are directly set up by random numbers, and the volume estimation example we give is of this type where the events are placing individual points. It is no trick to figure out whether a given point `(p.x,p.y,p.z)` lies within a radius `sphere.r` of a point `(sphere.x,sphere.y,sphere.x)`, so generating a large number of randomly-placed points and counting those that lie within the sphere's radius from the sphere's center for some one or more spheres is not difficult.

Other kinds of Monte Carlo modeling might be a little more challenging. There is a famous experiment that estimates the value of p, for example; called the Bouffon needle experiment, it consists of drawing a number of parallel lines on a sheet of paper exactly as far apart as the length of a needle, and then dropping a large number of these needles on the sheet. The proportion of needles that cross one of the lines is an approximation of $2/\pi$. Simulating this with computer graphics is straightforward: you generate one random point as one end of the needle, generate a random angle (number between 0 and 2π) and place a second point one unit distant from the first along that angle, and compare the values of the endpoints to see whether the "needle" crossed the "line." And, of course, you can draw the needles and the lines as you go so that the viewer can watch the experiment proceed. It might be a pleasant diversion to code this up some rainy day!

4D graphing

A 3D scalar field is a more difficult object to display because it has a 3D domain and 1D range, so we are working at the 4D level. We saw two different ways of handling this case, and of course there would be many more besides these. The code for these two approaches is fairly straightforward. For the isosurface approach, we divide the volume into a number of voxels and evaluate the scalar field function at each of the eight vertices of the voxel. If the function passes through the fixed value that defines the isosurface, that voxel contains part of the surface and so we draw a sphere at that location. We identify whether the function passes through the fixed value by subtracting that value from the value of the function at each vertex and then multiplying these differences; if the sign of that product is negative for any of the edges, the fixed value is crossed. So the code consists of a triple-nested loop with a number of tests, and if the test is positive, we draw a sphere, as follows:

```
for ( i=0; i<XSIZE; i++ )
   for (j=0; j<YSIZE; j++)
      for ( k=0; k<ZSIZE; k++ ) {
         x = XX(i); x1 = XX(i+1);
         y = YY(j); y1 = YY(j+1);
         z = ZZ(k); z1 = ZZ(k+1);
         p1 = f( x, y, z); p2 = f( x, y,z1);
         p3 = f(x1, y,z1); p4 = f(x1, y, z);
         p5 = f( x,y1, z); p6 = f( x,y1,z1);
         p7 = f(x1,y1,z1); p8 = f(x1,y1, z);
         if (((p1-C)*(p2-C)<0.0) || ((p2-C)*(p3-C)<0.0) ||
             ((p3-C)*(p4-C)<0.0) || ((p1-C)*(p4-C)<0.0) ||
             ((p1-C)*(p5-C)<0.0) || ((p2-C)*(p6-C)<0.0) ||
             ((p3-C)*(p7-C)<0.0) || ((p4-C)*(p8-C)<0.0) ||
             ((p5-C)*(p6-C)<0.0) || ((p6-C)*(p7-C)<0.0) ||
```

```
            ((p7-C)*(p8-C)<0.0) || ((p5-C)*(p8-C)<0.0) ) {
               drawSphere(x,y,z,rad);
            }
      }
```

For the cutting plane display, we simply define a plane in the space and iterate across it in the same way we would for a 2D scalar field. That is, we use the 3D grid in the space put a 2D mesh on the two remaining variables, calculate the value of the function at the midpoint of each rectangle in the mesh, and draw the mesh rectangles in 3D space in a color determined by the function value; because this is so much like the 2D scalar field case, we will not include code for it here. It is straightforward to use interaction techniques to change the coordinate axis across which we are cutting or to change the value on that axis that defines the cut. It would be possible, though a little more complicated, to define other planes to cut the space, but we have not done that.

When we consider 2D vector fields on a 2D domain, we again have four dimensions and we have a choice as to our display. We obviously cannot display an actual 2D vector at each point of the domain, because that would make it essentially impossible to find any single vector. However, if we know that the vector field is relatively smooth, we can display the result vector at selected points in the domain, giving rise to an image with vectors of various lengths and directions. Here we are asking the viewer to understand the results and integrate an image that could have (or could not have, depending on the vector field) overlapping result vectors. This is not a bad approach, but it would take some care to make it work.

We have chosen a slightly different approach in showing the magnitude and direction of the result vectors separately. The magnitude is simply a scalar field, and we have seen how we can readily display it through techniques such as pseudocolor ramps. With the magnitude removed, the vector's direction can be displayed as a unit vector in the appropriate direction, which shows how the directions are distributed across the domain. Together we get a fairly simple display, but we need to understand that it might not be immediately obvious to a user what the display is saying, because the color and direction are disconnected. These vectors are drawn for the middle point in each 10x10 block of the grid in the scalar field after the scalar field itself has been drawn, and we draw each of the vectors in cyan. We assume that we have calculated the low and high x and y values for each of the grid rectangles, and have calculated the vectors for the midpoint of the rectangle. The code below sketches how we could create the vector portion displays shown for this section.

```
if ((i%10==5) && (j%10==5)) { // middle of every 10th cell
x = 0.5*(XX(i)+XX(i+1));
y = 0.5*(YY(j)+YY(j+1));
len = 5.0 * sqrt(vector[0]*vector[0]+vector[1]*vector[1]);
glBegin(GL_LINE_STRIP);
   glColor4f(0.0, 1.0, 1.0, 1.0);
   glVertex3f(x,y,EPSILON); //so the vector is above the surface
   glVertex3f(x+vector[0]/len, y+vector[1]/len, EPSILON);
glEnd();
```

Higher dimensional graphing

When we get into any higher-dimensional graphics, we must be very careful to keep the image clear and focused, because it can easily become confused. With more information in the other dimensions that we cannot readily show, you need to plan what data to present and how to present it, or you need to plan how you can allow your viewer to make these choices.

When we talk about vector fields on domains where the dimension is larger than two, we have the problems described above about showing too much information, as well as problems caused by

projections hiding some of the information. It is extremely important to allow the viewer to move around the data (or alternately to allow the viewer to move the data volume around) so that it can be seen from all angles. It is also important to show only selected information so that the viewer can get the sense of the data instead of everything at once. For example, when we show vectors in 3D space as in the display of a magnetic field, we cannot show a separate vector for every point in the space because the vectors would cover each other completely. Instead, we would use a technique similar to the vector display above and show only a relatively few vectors in the space. By placing these in a regular pattern in the space, then, we would show the shape of the vector field rather than the complete field. Code for this would be straightforward, as this pseudocode shows:

```
set the color for the vectors
for i
   for j
      for k{
         calculate coordinates of the i,j,k-th point
         calculate vector from magnetic field function for point
         begin lines
            set the point
            set the offset from that point by the vector
         end lines
      }
```

We do not pursue all the possible directions in higher-dimensional graphing here, however. As we saw in the chapter on visual communication, there are ways to use color, shape, and other clues to indicate higher-dimensional information. Some of these work better for nominal data (e.g. shapes), and some for ordinal data (e.g. colors), but considerable creativity can be called for when you need to go very far in this direction.

Data-driven graphics

Displaying data inherently has some of the same problems that any work with data will have. Data comes from many sources and is not always accurate, is not always exactly comparable with other data values (e.g. data may be taken at different times of the day), and may not measure precisely what we would like to present. We also may have some problems because the data may not be collected in ways that make it easy to present. But as long as there is ...

Credits

A number of colleagues have helped with graphical concepts, with scientific principles, or with models of the scientific issues. We apologize in advance for those we may miss, but we want to thank Michael J. Bailey of the San Diego Supercomputer Center (SDSC) for many fruitful discussions across all of these topics, and with Kris Stewart of San Diego State University (SDSU), Angela Shiflet of Wofford College, Rozeanne Steckler and Kim Baldridge of SDSC, and Ian Littlewood of California State University Stanislaus for their contributions in one or more of these areas.

Chapter 8: The Rendering Pipeline

Prerequisites

An understanding of graphics primitives and the graphics pipeline that will enable the student to see how the primitives are handled by the pipeline operations discussed in this chapter.

Introduction

In an earlier chapter we saw the outline of the graphics pipeline at a rather high level, and then we described how the API operations transform information in model coordinates to screen coordinates. With these screen coordinates, it is still necessary to carry out a number of operations to create the actual image you see on your screen or other output device. These operations are carried out in several ways, depending on the graphics system used, but in general they also have a pipeline structure that we will call the rendering pipeline because it creates the rendered image from the geometry of the graphics pipeline.

We should be careful to point out that the rendering pipeline as we will describe it applies mainly to polygon-based graphics systems that are rendered by processing each polygon through operations that develop its appearance as they render it into the scene. Not all graphics systems work this way. A ray-tracing system will generate a ray (or a set of rays) for each pixel in the display system and will calculate the intersection of the ray with the nearest object in the scene, and will then calculate the visible appearance of that intersection from properties of the object or from operations based on optical properties of the object. The rendering process here is simply the appearance calculation, not of a whole polygon, even if the scene should be made up of polygons. Thus ray tracing has no rendering pipeline in the sense we describe in this chapter.

In this chapter we will look at the rendering pipeline for graphics systems based on per-polygon properties in some detail, describing the various operations that must be performed in rendering an image, and eventually focusing on the implementation of the pipeline in the OpenGL system.

The pipeline

When we begin to render the actual scene, we have only a few pieces of information to work from. We have the fundamental structure of the various pieces of the scene (such as triangles, rects, bitmaps, texture maps, lights, or clipping planes). We have the coordinates of each point that describes the geometry along with additional information for each point such as the color of the point, the normal at the point, the texture name and texture coordinates for the point, and the like. We also have the basic information that describes the scene, such as whether or not we have enabled depth buffering, smooth shading, lighting, fog, or other operations. The rendering task is to take all of this data, some of which will change from object to object in the scene, and to create the image that it describes.

This process takes place in several stages. In one, the graphics pipeline applies the instancing transformations and viewing transformation that you define to create 3D eye space vertex data from your original model data, creating a complete representation of the properties of the vertex. In a second, the vertex data for each polygon in your scene is interpolated to raster information to define the endpoints of scanlines so that the pixels in the polygons may be displayed. In another, color is defined from color data or texture data is applied to the pixels of the image as they are rendered. In yet another, the data for each pixel that is generated from the simple rendering is modified to apply effects such as depth testing, clipping, fog, or color blending. Overall, these processes provide the computations that make high-quality visual representations of the model you have defined with the image properties you have specified.

We have already seen the effect of the graphics pipeline and have alluded to it above. When we discussed it early in these notes, we focused on transformation operations on the vertices of your models that transformed the vertices from model space into 2D screen space. The eventual data structure that holds vertex information, though, holds a complete description of the vertex and is much richer than just the 2D x- and y-coordinates of a screen point. It also holds a depth value, the depth of the pixel in the original model space, which is needed for accurate polygon interpolation; color information, needed for determining the color for a polygon or for simple color interpolation; texture coordinates of the vertex, needed for texture mapping; and other information that depends on the graphics API used. For example, you would want to include the vertex normal in world space if you were using Phong shading for your lighting model.

The rendering pipeline, then, starts with the original modeling data (an original model vertex, data defined for that vertex, and the transformation to be applied to that vertex) and creates the screen representation of each vertex. That screen vertex is part of the definition of a polygon, but to complete the definition of a polygon the pipeline must gather the information on all the polygon's vertices. Once all the vertices of the polygon are present, the polygon can begin to be rendered, which involves defining the properties of each pixel that makes up the polygon and writing those pixels which are visible to the graphics output buffer.

We will assume that the graphics hardware you will use is scanline-oriented, or creates its image a line at a time as shown in Figure 8.1. A *scanline* is the set of pixels on your display device that have the same value of y; it is one horizontal row. The set of pixels in a polygon on one scanline is called a *fragment*. Rendering the polygon requires that you define all the fragments that comprise the polygon and determine the properties of all the pixels in each fragment. Notice that on a convex polygon, each scanline will meet the polygon in a single fragment, while on a non-convex polygon there may be more than one line segment on a scanline. This is why most graphics APIs only work with convex polygons and require you to break up a non-convex polygon into convex parts before it is displayed.

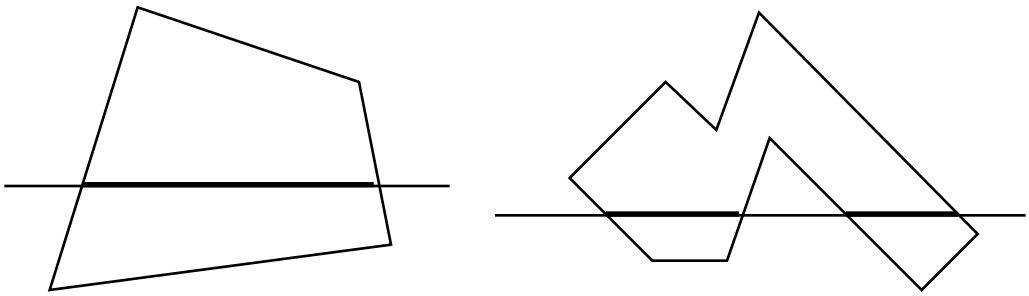


Figure 8.1: a scanline on a convex polygon (left) and general polygon (right)

Once you have defined the vertices of the polygon in screen space, the next step in the rendering pipeline is interpolating the polygon vertices to define the points on the edges of the polygon that are the endpoints of the scanline segments so you can process these segments and write them to the frame buffer. Here you will use either linear or perspective-corrected interpolation to determine the coordinates of the original point on the polygon that would be projected to the screen point that lies on the desired scan line. The perspective-corrected interpolation affects the depth, texture coordinates, and possibly other data for the interpolated vertices. This is discussed in more depth in the later chapter on texture mapping.

Once you have the scanline endpoints (and their data) calculated, you can begin to create the pixels on the scanline between the endpoints, filling this interval of the polygon. Again, you must interpolate the data between the endpoints, and again, some perspective correction may be applied.

Now, however, you have actual colors or actual texture coordinates of each individual pixel so you may determine the color each pixel is to be given. However, not all pixels are actually written to the output buffer because there may be depth testing or clipping applied to the scene, so there are now several tests that must be applied to each pixel before actually writing it out. If depth testing is being done, then the pixel will only be written if the depth is less than the depth at this pixel in the depth buffer, and in that case the depth buffer will be modified to show this pixel's depth. If other clipping planes have been enabled, the original coordinates of the pixel will be recalculated and compared with the clipping plane, and the pixel will or will not be written depending on the result. If there is an alpha channel with the pixel's color and the color is visible, color blending will be applied with the pixel color and the color in the image buffer before the pixel is written. If there is a fog effect, then the fog calculations will be done, depending on the pixel depth and the fog parameters, before the pixel is written. This set of pixel-by-pixel operations can be expensive, so most of it can be enabled or disabled, depending on your needs.

In addition to this set of operations for displaying pixels, there are also operations needed to create the texture information to be applied to the pixels. Your texture map may come from a file, a computation, or a saved piece of screen memory, but it must be made into the internal format needed by your API. This will usually be an array of color values but may be in any of several internal formats. The indices in the array will be the texture coordinates used by your model, and the texture coordinates that are calculated for individual pixels may well not be integers so there will need to be some algorithmic selection process to get the appropriate color from the texture map for each pixel in the pixel pipeline above.

The rendering pipeline for OpenGL

The OpenGL system is defined in terms of the processing described by Figure 8.2, which outlines the overall system structure. In this figure, system input comes from the OpenGL information handled by the processor and the output is finished pixels in the frame buffer. The input information consists of geometric vertex information, transformation information that goes to the evaluator, and texture information that goes through pixel operations into the texture memory. The details of many of these operations are controlled by system parameters that you set with the

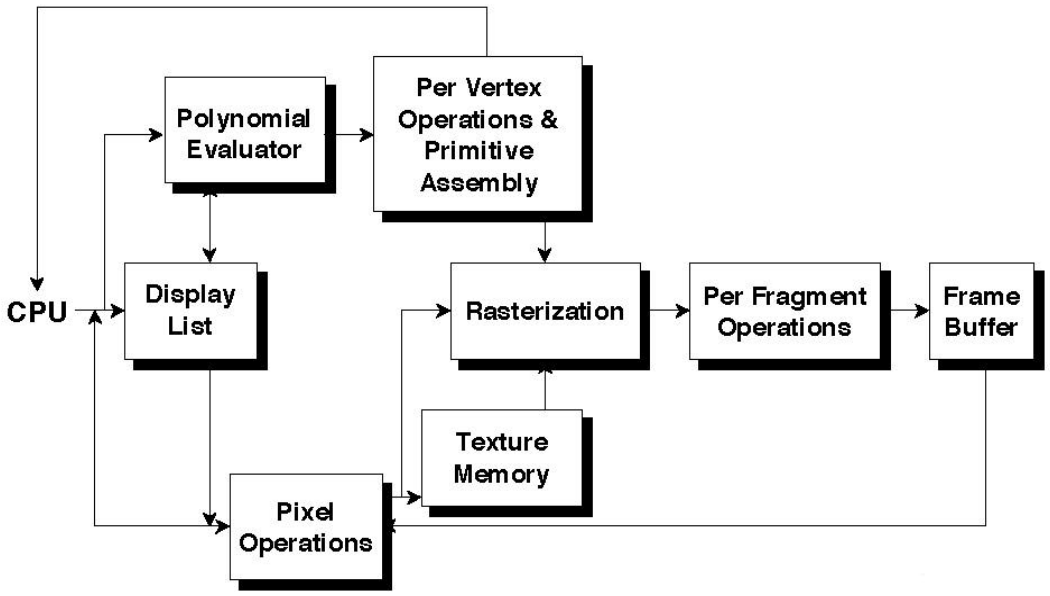


Figure 8.2: the OpenGL system model

`glEnable` function and are retained as state in the system. We will outline the various stages of the system's operations to understand how your geometry specification is turned into the image in the frame buffer.

Let us begin with a simple polygon in immediate-mode operation. The 3D vertex geometry that you specify is passed from the CPU and is then forwarded to the per-vertex operations. If you have specified lighting operations, the color data for each vertex is calculated based on the geometry of the 3D eye coordinate system. Here the modelview matrix, which includes both the model transformation and the viewing transformation in OpenGL, is applied to the vertices in modeling coordinates to transform them into 3D eye coordinates, and clipping is done against the edges of the view volume or against other planes if enabled. Next the projection transformation is applied to transform the vertices into 2D eye coordinates. The result is the transformed vertex (along with the other information on the vertex that has been retained through this process), ready for primitive assembly and rasterization.

If you are compiling display lists instead of working in immediate mode, then instead of passing on the vertex data and OpenGL operations into the polygon evaluator to do the projections, the data goes into display list memory for later use. When the display list is executed, the operations continue essentially the same as they would had they been passed in immediate mode, except that there is no function call overhead and there may be some optimization done on the data as it is put into the display list.

From this point, then, the vertices of the completed primitives go into the rasterization stage. This stage applies the interpolation and scanline processing described above. You can control whether a perspective-corrected interpolation is performed by using the `glHint` function with parameter `GL_PERSPECTIVE_HINT` as described in the chapter on texture mapping. As the individual pixels are computed, appropriate color or texture data is computed for each depending on your specifications, and the resulting scan-line data is ready to go on the per fragment operations.

You may not have noticed the feeback line from the per-pixel operations to the CPU, but it is very important. This is the mechanism that supports pick and selection operations that we will discuss in a later chapter. Here the connection allows the system to note that a given pixel is involved in creating a graphics primitive object and that fact can be noted in a data structure that can be returned to the CPU and thus to the application program. By processing the data structure, the application can see what primitives lie at the given point and operate accordingly.

While we have been discussing the actions on vertex points, there are other operations in the OpenGL system. We briefly mentioned the use of the polynomial evaluator; this comes into play when we are dealing with splines and define evaluators based on a set of control points. These evaluators may be used for geometry or for a number of other graphic components, and here is where the polynomial produced by the evaluator is handled and its results are made available to the system for use. This is discussed in the chapter on spline modeling below.

Texture mapping in the rendering pipeline

Texture mapping involves other parts of the rendering system. Here a texture map is created from reading a file or from applying pixel operations to data from the frame buffer or other sources. This texture map is the source of the texture data for rasterization. Information in the texture map, which is simply an array in memory, is translated into information in texture memory that can be used for texture mapping. The arrow from the frame buffer back to the pixel operations indicates that we can take information from the frame buffer and write it into another part of the frame buffer; the arrow from the frame buffer to texture memory indicates that we can even make it into a texture map itself. This is described by Figure 8.3.

Here we see that the contents of texture memory can come from the CPU, where they are translated into array form after being read from a file. Because OpenGL does not know about file formats, it can be necessary to decode data from the usual graphics file formats (see [Murray & vanRyper], for example) into array form. However, the texture memory can also be filled by copying contents from the frame buffer with the `glCopyTex*Image(...)` function or by performing other pixel-level operations. This can allow you to create interesting textures, even if your version of OpenGL does not support multitexturing.

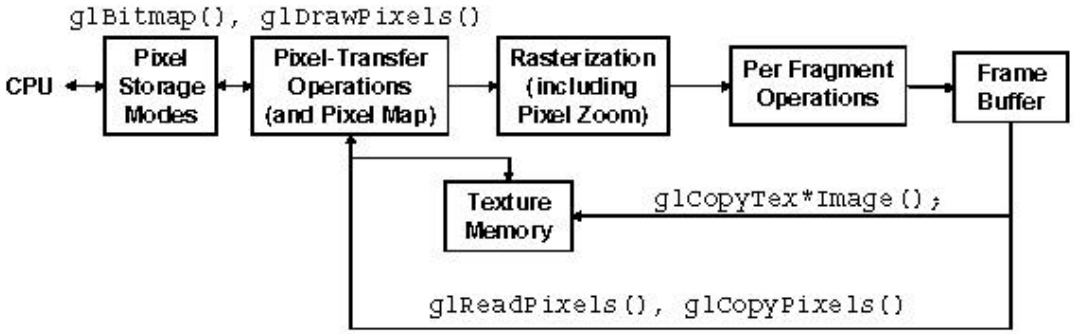


Figure 8.3: processing for texture maps

It is rare for an individual pixel to have a texture coordinate that exactly matches the indices of a texture point. Instead of integer texture coordinates, the pixel will have real-valued texture coordinates and a calculation will need to be done to determine how to treat the texture data for the pixel. This can involve choosing the nearest texture point or creating a linear combination of the adjoining points, with the choice defined by the `glTexParameter` function as described in the texture mapping chapter.

Per-fragment operations

Much of the power of the OpenGL system lies in its treatment of the fragments, or small sets of scanline data, that are computed by the rasterization process. The fragment operations follow a sub-pipeline that is described in Figure 8.4. Some of the fragment operations probably fall under

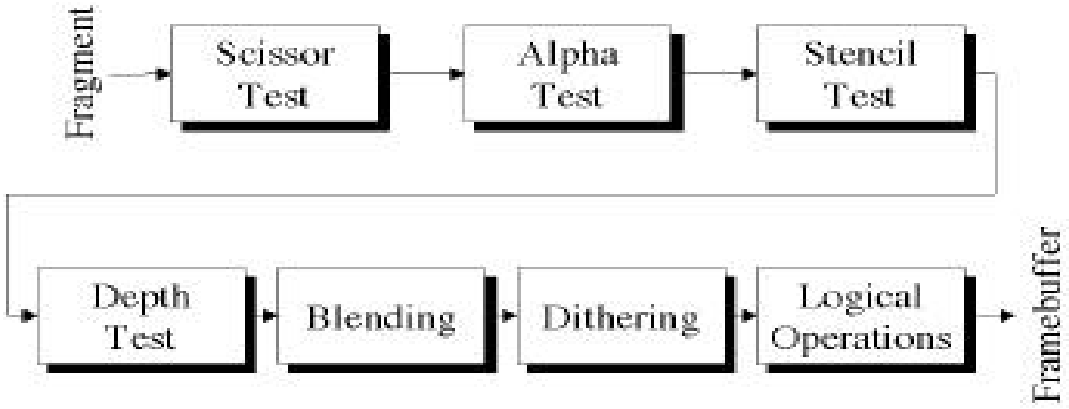


Figure 8.4: details of fragment processing

the heading of advanced OpenGL programming and we will not cover them in depth, although many of them will be covered in later chapters. Most of them are operations that must be enabled

(for example, with `glEnable(GL_SCISSOR_TEST)`, `glEnable(GL_STENCIL_TEST)`, or the like) and some require particular capabilities of your graphics system that may not always be present. If you are interested in any details that aren't covered adequately here, we would direct you to the OpenGL manuals or an advanced tutorial for more information.

The first fragment operation is a scissor test that lets you apply additional clipping to a rectangular bounding box defined by `glScissor(...)`, and the second operation allows you to use a test against a pixel's alpha value to create a mask for textures, defined by `glAlphaFunc(...)`. The next operation applies a stencil test, which is much like the alpha test except that it creates a mask based on values in a stencil buffer. The stencil operations are based on a stencil mask that you can draw to with normal OpenGL operations and that is used to choose whether or not to eliminate a pixel from a fragment when it is drawn. The stencil test is based on a comparison of the value in the stencil buffer and a reference value, and each pixel in the fragment is either kept or replaced by a value that you can set. The key functions for stencil testing are `glStencilFunc(…)` to set the test function, `glStencilMask(…)` to control writing to the stencil buffer, and `glStencilOp(…)` to specify the actions for the stencil test.

The next set of operations are more familiar. They begin with the depth test that compares the depth of a pixel with the depth of the corresponding point in the depth buffer and accepts or rejects the pixel, updating the depth buffer if the pixel is accepted. Following this is the blending operation that blends the color of the pixel with the color of the frame buffer as specified in the blending function and as determined by the alpha value of the pixel. This operation also supports fog, because fog is primarily a blending of the pixel color with the fog color. The dithering operation allows you to create the appearance of more colors than your graphics system has by using a combination of nearby pixels of different colors that average out to the desired color. Finally, the logical operations allow you to specify how the pixels in the fragment are combined with pixels in the frame buffer. This series of tests determines whether a fragment will be visible and, if it is, how it will be treated as it moves to determine a pixel in the frame buffer.

Some extensions to OpenGL

In this section we'll discuss programmable vertex and fragment operations and the idea of a shading language, with the goal of giving you some background on these ideas. We expect that future versions of OpenGL, or at least generally accepted extensions, will allow this kind of programmable operations.

In standard OpenGL, when a vertex comes into the rendering pipeline we know much more about it than just its coordinates. We also know its color (whether determined by a lighting model or by simply setting the color), and perhaps its texture coordinate. There is no reason why we cannot define much more than this about a vertex, however. We can also store displacement vectors, up to eight multitexture coordinates, and particular transformations. Vertices can even store addresses of programs that can compute shape, color, anisotropic shading by computing lighting-oriented normals instead of geometric normals, or bump maps. Graphics cards are beginning to include quite a bit of per-vertex programmability with 16 or more 4D real vectors per vertex to hold additional data, although each card will have a distinct instruction set that is oriented to its particular architecture.

Besides this kind of program that can be attached to each vertex, however, we can also apply other techniques to per-fragment operations than are included in the fragment processing described above. There are programmable dot product operations in some graphics cards, modeled on the idea of texture combining operations, that will allow you to apply additional kinds of operations for processing fragments. The end result can be thought of as a programmable rendering pipeline, with three programmable stages: group processing, vertex processing, and fragment processing.

Two of these are familiar from the OpenGL rendering pipeline in Figure 8.2, with group processing representing operations on collections of vertices instead of individual vertices for efficiency. This programmable pipeline is shown in Figure 8.5.

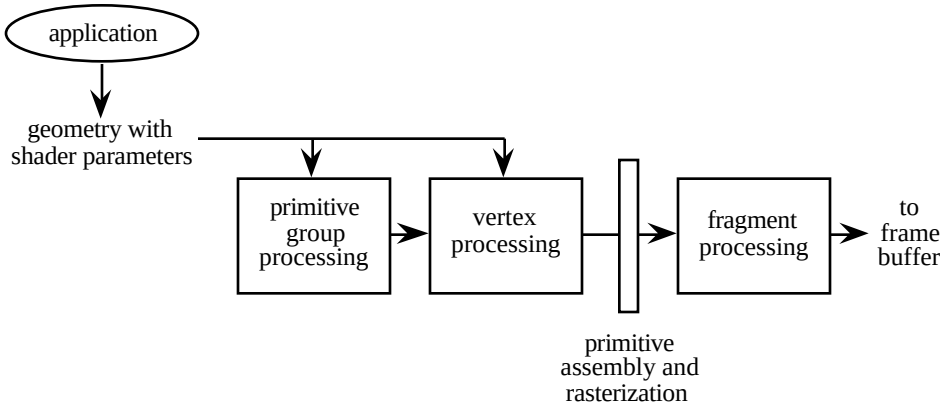


Figure 8.5: Programmable pipeline with three programmable stages

These give us a good idea of some kinds of developments we should expect to see in future versions (or extensions) of OpenGL or other graphics APIs. We should expect to be able to provide a program with each vertex to compute vertex properties such as we described above. For compatibility with a wide range of hardware, the language of such a program will probably be independent of the particular graphics card, and the graphics API will provide a way to either compile or interpret the language into the specific operations needed for the card. We expect that the language for per-vertex shaders will probably look a lot like RenderMan™, because this language is already designed for writing shader operations and is rather familiar to many in the computer graphics community. This language should provide at least surface shaders (programs that compute an RGBA color for the frame buffer) and light shaders (programs that compute light information for use with the surface shader), and likely other kinds of shader operations as well. It will be interesting to see what this does for advanced programming with graphics APIs.

An implementation of the rendering pipeline in a graphics card

The system described above is very general and describes the behavior required to implement the OpenGL processes. In practice, the system is implemented in many ways, and the diagram in Figure 8.6 shows the implementation in a typical fairly simple OpenGL-compatible graphics card. The pipeline processor carries out the geometry processing and produces the fully-developed screen pixels from the original modeling space vertices as described above. The texture memory is self-describing and holds the texture map after it has been decoded by the CPU. The rasterizer handles both the rasterization operations and the per-fragment operations above, and the Z-buffer and double-buffered framebuffer hold the input and output data for some of these operations. The cursor is handled separately because it needs to move independently of the frame buffer contents, and the video driver converts both the frame buffer content and the other inputs (cursor, video) to the format that drives the actual monitor display mechanism. This kind of straightforward mapping of API functionality to hardware functionality is one of the reasons that OpenGL has become an important part of current graphics applications—it provides good performance and good price points to the marketplace.

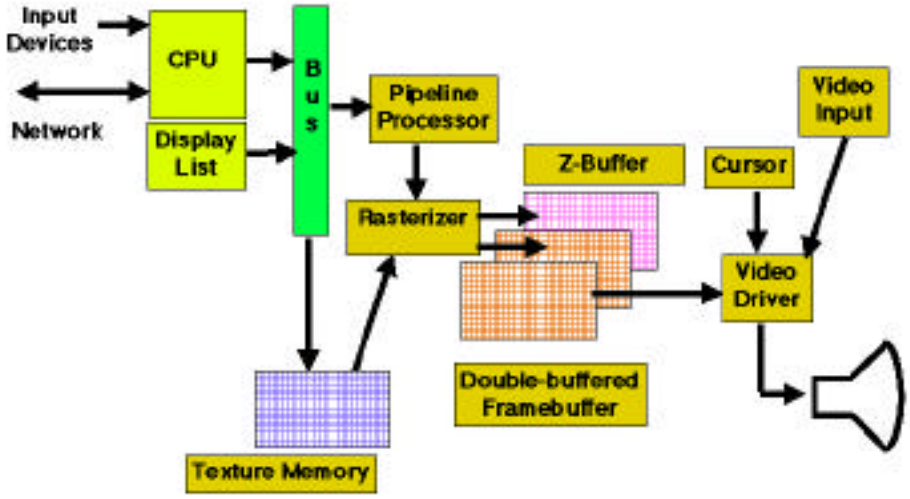


Figure 8.6: an implementation of the OpenGL system in a typical graphics card

The rasterization process

In our discussion of the graphics pipeline, we saw that the rasterization process plays a key role. Polygons come into the process in terms of their vertices, and the vertex geometry is translated into scanline-oriented fragments for further detailed processing, whether by texture mapping or by the various per-fragment operations. The end result is the polygon as it is meant to be displayed by your program. Rasterization is carried out by the OpenGL system so it is not necessary to understand it in order to do graphics programming, but there are details of the process that will help you understand some of the basic concepts of computer graphics. Here we will describe the rasterization process in some detail.

First, let's recall the information that is present at each vertex in the geometry as it gets to the rendering pipeline. We have the 2D screen coordinates of the vertex, calculated by projecting the vertex from 3D eye space to 2D eye space and then mapping that space to screen space. We have the z-value of the vertex in 3D eye space, because we do not need to change it for the screen display but we do need it for some computations; this z-value may be in its original terms, or it may have been converted to a more convenient form such as the integer value having 0 at the front of the view volume and the largest integer at the back of the view volume, as OpenGL stores it. We have the color of the vertex, usually as an RGB triple, either given by the model or calculated from the lighting model. (If we were using Phong shading, instead of the color of the vertex we might have the normal at the vertex for later shading computations.) And we have the texture coordinates of the vertex. So each vertex carries a great deal more information than just its screen geometry.

As we go through the rasterization process, we must take the vertices from the geometry and scan-convert them—interpolate them and find the pixels the polygon will use for each scan line— to determine the total set of pixels that are displayed for the polygon. The scan conversion process operates first on each edge of the polygon to get the set of pixels that represent that edge. When this is done for all the edges, you will have all the pixels that bound the polygon. For a convex polygon, only two edges can intersect any one scan line, so you can organize the pixels into a set of pairs, one for each scan line. Each pair then determines a fragment by interpolating the pixels between them.

There are many algorithms for performing the rasterization itself, so we will choose the one that is probably the simplest: the DDA (Digital Differential Analyzer) algorithm. This algorithm takes the screen-space coordinates of the two endpoints of a line segment and uses the straightforward line equation and roundoff to calculate the pixels of the line segment on each scanline. Because each pixel's coordinates are integers and a line segment is continuous, we must realize that we will create an approximation of the line segment, not the exact segment; we will make this a best approximation by calculating the pixel on each scanline that is the closest to the real-valued point on that scanline. So we will create an aliased version of the line segment, but it will be the best alias we can create.

To begin, let's assume that our line segment has endpoint vertices (X1, Y1) and (X2, Y2) and let's label $\Delta X = X2 - X1$ and $\Delta Y = Y2 - Y1$. For the sake of convenience, we will assume that ΔX and ΔY are both positive; if they are not then you can adjust the algebraic sign in what we do below. Notice also that we can translate our line segment however we want, because once we calculate all the pixels for the line we can translate the entire line segment by translating each pixel of the segment. So we can assume that our line segment lies in the first quadrant.

Now the nature of the pixels for the line segment differs if $\Delta Y > \Delta X$ or $\Delta X > \Delta Y$. If $\Delta Y > \Delta X$, then there will be only one pixel on each scanline, while if $\Delta X > \Delta Y$, there will be only one pixel lying on any vertical line in screen space. We will develop our algorithm for the case $\Delta Y > \Delta X$, but you can exchange the X and Y terms in the algorithm to deal with the other case.

So for each scanline between Y1 and Y2, we will want to compute the pixel on the scanline that best represents the exact point on the line segment. We will begin with the equation of a line that calculates X as a function of Y:

$$X = X1 + ((Y - Y1)/(Y2 - Y1)) * (X2 - X1)$$

Here the term $(X2 - X1)/(Y2 - Y1)$ represents the slope of the line in terms of $\Delta X/\Delta Y$ instead of the more usual $\Delta Y/\Delta X$ because we are calculating the value of X as a function of Y. Once we have calculated the value of X for each (integer) scanline Y, then, we simply round that value of X to the nearest integer to calculate the pixel nearest the actual line. As pseudocode, this becomes the algorithm:

```
Input:  two screen points (X1,Y1) and (X2,Y2) with Y2 > Y1 and
        with (Y2-Y1)/(X2-X1) > 1
Output: set of pixels that represents the line segment between
        these points in screen space
for (int Y = Y1; Y < Y2; Y++) {
   // we do not include Y2 as discussed below
   float P = (Y-Y1)/(Y2-Y1);
   float X0 = X1 + P*(X2-X1);
   int X = round(X0);
   output point (X,Y);
}
```

The operation of this algorithm is illustrated in Figure 8.7, where we show a raster with two endpoints and then show how the DDA algorithm populates the scanlines between the endpoints. Note that a roundoff that rounds an X-value upwards will give you a pixel to the right of the actual line, but this is consistent with the relation between the pixels and the line given by the endpoints.

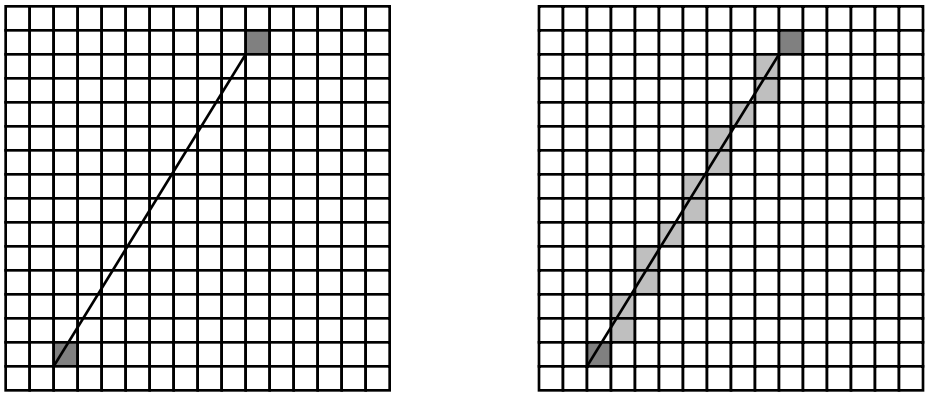


Figure 8.7: scan converting an edge with the DDA algorithm

Now when you scan-convert a complete polygon, you begin by scan-converting all the line segments that make up its boundaries. The boundary pixels are not written immediately to the frame buffer, however. You must save these pixels in some sort of pixel array from which you can later get them for further processing into the fragments that make up the polygon. A good mental model is to think of a 2D array, with one index being the scanline and the other representing the two pixels you might have for each scanline (recalling that for a convex polygon, each scanline will meet the polygon in zero or two points). You will write each pixel to the appropriate array, and as you do so you will sort each scanline's 2D array by the X-value of the pixels. Each of these 2D arrays, then, represents a fragment—a line segment within the polygon having a constant scanline value—for the polygon.

There are some details of handling these scanline fragments that you need to understand, because the process as we have defined it so far includes some ambiguous points. For example, we have not talked about the "fragment" you would get at the highest or lowest vertex in the polygon, where you would have the same pixel included twice. We have also not talked about the relation between this polygon and others with which it might share an edge; that edge should be part of one, but not both, of the polygons. If we would include it in both polygons, then the image we get would depend on the order in which the polygons are drawn, which would be a problem. In order to address these problems, we need to introduce a couple of conventions to creating fragments. First, we will assume that we will include a horizontal boundary fragment only if it represents the bottom of the polygon instead of the top. This is easily handled by including every pixel except the topmost pixel for any non-horizontal boundary. Second, we will include any left-hand boundary in a polygon but no right-hand boundary. This is also easily handled by defining the fragment for each scanline to include all pixels from, and including, the left-hand pixel up to, but not including, the right-hand pixel. Finally, we are handling all scanlines as fragments, so we do not process any horizontal edge for any polygon.

With the algorithm and conventions defined above, we are able to take any convex polygon and produce the set of fragments that present that polygon in the frame buffer. As we interpolate the vertices across scanlines, however, there are other things defined for each pixel that must also be interpolated, including color, depth, and texture coordinates. Some of these properties, such as color, are independent of the pixel depth, but others, such as texture coordinates and depth itself, are not. Any depth-independent property can be treated simply by interpolating it along each edge to get values for the endpoints of each fragment, and then interpolating it along the fragment when that is processed. This interpolation can be computed in exactly the same way as the DDA algorithm interpolates geometry.

But for depth-dependent properties, we do not have a linear relationship between the property and the pixel in the plane. As we interpolate linearly across the pixel coordinates in screen space, the actual points on the line segment that correspond to these pixels are not themselves distributed linearly, as we see in Figure 8.8, where the space between points at the top of the actual line segment is much larger than the space between points at the bottom. We must use a perspective correction in order to reconstruct the actual point in 3D eye space. Recalling from Chapter 2 that the perspective projection gets the 2D eye space coordinates of a vertex by dividing by the vertex's *z*-value, we need the actual depth value z to compute the original vertex coordinates. Once we calculate—or estimate, which is all we can do because of the aliased nature of pixel coordinates—the original depth value, we can estimate the original vertex coordinates and then use simple geometric principles to estimate the actual texture coordinates.

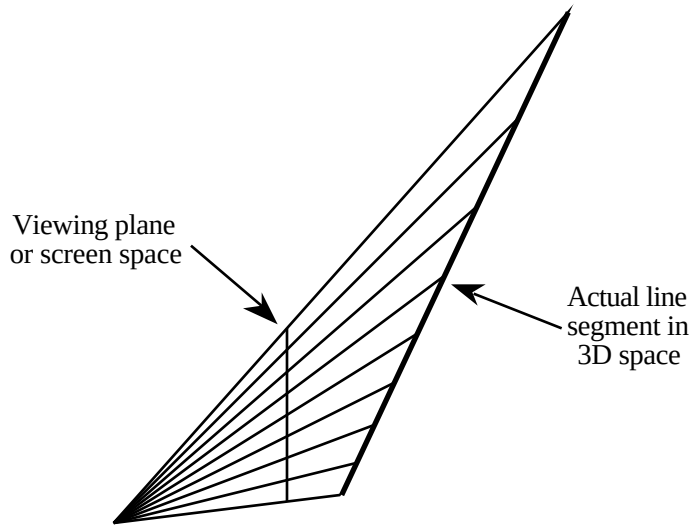


Figure 8.8: The distribution of points on the original edge that correspond to a linear sequence of pixels

To interpolate the z-values, we must recognize that we are interpolating points that have been transformed by a perspective transformation. If the original point in 3D eye space is (x,y,z) and the point in 2D eye space is (X,Y), that transformation is given by $X = x/z$ and $Y = y/z$. Now if we are interpolating, say, $X1$ and $X2$, then we are interpolating $x1/z1$ and $x2/z2$, so that to interpolate the key values, we must interpolate $1/z1$ and $1/z2$ to get our estimate of z for our interpolated point. If we consider the point $X = (1-t)*X1+t*X2$, then the corresponding z would be $(1-t)/z1 + t/z2 = ((1-t)*z2+t*z1)/(z1*z2)$. We would then reconstruct the original point in 3D eye space by multiplying the X and Y values of the interpolated point by this estimated z value.

We have discussed scanline interpolation in terms of a very simple, but not very fast, algorithm. In most cases, the scanline interpolation is done by a somewhat more sophisticated process. One such process is the Bresenham algorithm. This depends on managing an error value and deciding what pixels to light based on that value. For the usual basic case, we assume that the line we are interpolating has a slope no larger than one, and that the left-hand vertex is the (0, 0) pixel. Such lines are said to lie in the first octant of the plane in standard position. For such a line, we will set a single pixel for each value of X in screen coordinates, and the question for any other pixel is simply whether the new pixel will be alongside the previous pixel (have the same Y value) or one unit higher than the previous pixel, and this is what the Bresenham algorithm decides.

The algorithm takes as input two vertices, (X0,Y0) and (X1,Y1) with X0 < X1 and Y0 < Y1, and we compute the two total distance terms DX = (X1 – X0) and DY = (Y1 – Y0). We want to set up a simple way to decide for any value of X whether the value of Y for that X is the same as the value of Y for X–1 or one larger than the value of Y at X–1.

We begin with the first vertex, which we be at the lower left of the space the line segment will live in, and we ask where the pixel will be for X0+1. In the leftmost part of Figure 8.9 we see the setup for this question, which really asks whether the actual line will have a Y-value larger than Y0+.5 for X=X0+1. This can be rephrased as asking whether (DY/DX) > 1/2, or whether 2*DY>DX. This gives us an initial decision term P = 2*DY–DX, along with the decision logic that says that Y increases by one if P>0 and does not increase if P<0.

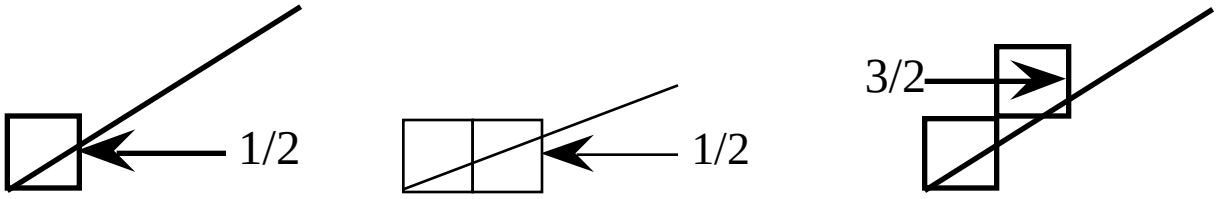


Figure 8.9: moving from one pixel to the next and considering the decision term

With the first new vertex of the line settled, let's now look at the second vertex. If we did not change Y for the first vertex, we find ourselves in the situation of the middle part of Figure 8.9. In this case, the decision for the second vertex is whether 2*(DY/DX) > 1/2. We calculate this out quickly as 4*DY > DX, or 4*DY–DX > 0. But this decision term can be written in terms of the previous decision term P as P+2*DY > 0. This case is, in fact, general and so we name the update term C1 = 2*DY and write the general operation: if P<0, then we create a new value of the decision variable P by P = P + C1.

But if we did change Y for the first vertex, we find ourselves in the situation of the right-hand part of Figure 8.9. Here the decision for the second vertex is whether 2*(DY/DX)–1 > 1/2. We again calculate this and get 2*DY > 3*DX/2, or 4*DY – 3*DX > 0. But again using the previous value of the decision term, we see that we now have P + 2*(DY–DX) > 0. Again, this case is general and we name the update term C2 = 2*(DY–DX) and write the general operation: if P>0, then we create a new value of the decision variable by P = P+C2.

The process of defining an initial value of the decision variable, making a decision about the next pixel, and then updating the decision variable depending on the last decision, is then carried out from the first pixel in the line to the last. Written as a code fragment, this first-octant version of the Bresenham algorithm implements the discussion above and looks like this:

```
bx = x0;
by = y0;
dx = (x1 - x0);
dy = (y1 - y0);
p = 2 * dy - dx;
c1 = 2 * dy;
c2 = 2 * (dy - dx);
while ( bx != x1 ){
   bx = bx + 1;
   if ( p < 0 ) p = p + c1; // no change in by
   else {
      p = p + c2;
      by = by + 1;
   }
setpixel( bx, by);
```

This algorithm can readily be made to interpolate not only lines in the first octant but lines in any direction, and can be readily adapted to interpolate also any property that is not depth dependent or, if you choose the property to be the reciprocal of the depth, can interpolate these values so you can approximate the actual depth of the pixel. A full writeup of the algorithm is provided in the accompanying materials.

Chapter 9: Lighting and Shading

Prerequisites

An understanding of color at the level of the discussion of the chapter on color in these notes, an observation of the way lights work in creating the images of the world around you, and an understanding of the concept of color, of polygons, and of intepolation across a polygon.

Lighting

There are two ways to think of how we see things. The first is that things have an intrinsic color and we simply see the color that they are. The color is set by defining a color in RGB space and simply instructing the graphics system to draw everything with that color until a new color is chosen. This approach is synthetic and somewhat simplistic, but it's very easy to create images this way because we must simply set a color when we define and draw our geometry. In fact, when objects don't have an intrinsic color but we use their color to represent something else, as we did sometimes in the science examples in an earlier chapter, then this is an appropriate approach to color for images.

However, it's clear that things in the real world don't simply have a color that you see; the color you see is strongly influenced by the lighting conditions in which you see them, so it's inaccurate not to take the light into account in presenting a scene. So the second way to think of how we see things is to realize that we see things because of light that reaches us after a physical interaction between light in the world and the objects we see. Light sources emit energy that reaches the objects and the energy is then re-sent to us by reflections that involve both the color of the light and the physical properties of the objects. In computer graphics, this approach to lighting that considers light that comes only from lights in the scene is called the *local illumination model*. It is handled by having the graphics programmer define the lights in the scene. In this chapter we will discuss how this approach to light allows us to model how we see the world, and how computer graphics distills that into fairly simple definitions and relatively straightforward computations to produce images that seem to have a relationship to the lights and materials present in the scene.

A first step in developing graphics with the local illumination model depends on modeling the nature of light. In the simplified but workable model of light that we will use, there are three fundamental components of light: the ambient, diffuse, and specular components. We can think of each of these as follows:

- ambient: light that is present in the scene because of the overall illumination in the space. This can include light that has bounced off objects in the space and that is thus independent of any particular light.
- diffuse: light that comes directly from a particular light source to an object, where it is then sent directly to the viewer. Normal diffuse light comes from an object that reflects a subset of the wavelengths it receives and that depends on the material that makes up an object, with the effect of creating the color of the object.
- specular: light that comes directly from a particular light source to an object, where it is then reflected directly to the viewer because the object reflects the light without interacting with it and giving the light the color of the object. This light is generally the color of the light source, not the object that reflects the light.

These three components are computed based on properties of the lights, properties of the materials that are in your scene, and the geometric relationship between the lights and the objects. All three of these components contribute to the overall light as defined by the RGB light components, and the graphics system models this by applying them to the RGB components separately. The sum of these three light components is the light that is actually seen from an object.

Just as the light is modeled to create an image, the materials of the objects that make up the scene are modeled in terms of how they contribute to the light. Each object will have a set of properties that defines how it responds to each of the three components of light. The ambient property will define the behavior (essentially, the color) of the object in ambient light, the diffuse property in diffuse light, and the specular property in specular light. As we noted above, our simple models of realistic lighting tend to assume that objects behave the same in ambient and diffuse light, and that objects simply take on the light color in specular light, but because of the kind of calculations that are done to display a lighted image, it is as easy to treat each of the light and material properties separately.

So in order to use lights in a scene, you must define your lights in terms of the three kinds of color they provide, and you must define objects not in simple terms of their color, but in terms of their material properties. This will be different from the process we saw in the earlier module on color and will require more thought as you model the objects in your scenes, but the changes will be something you can handle without difficulty. Graphics APIs have their own ways to specify the three components of light and the three components of materials, and we will also discuss this below when we talk about implementing lighting for your work.

Definitions

Ambient, diffuse, and specular light

Ambient light is light that comes from no apparent source but is simply present in a scene. This is the light you would find in portions of a scene that are not in direct light from any of the lights in the scene, such as the light on the underside of an object, for example. Ambient light can come from each individual light source, as well as from an overall ambient light value, and you should plan for each individual light to contribute to the overall brightness of a scene by contributing something to the ambient portion of the light. The amount of diffuse light reflected by an object from a single light is given simply by $A = L_A \quad C_A$ for a constant C_A that depends on the material of the object and the ambient light L_A present in the scene, where the light L_A and constant C_A are to be thought of as RGB triples, not simple constants, and the calculation is to yield another RGB value. If you have multiple lights, the total diffuse light is $A = \sum L_A \quad C_A$, where the sum is over all the lights. If you want to emphasize the effect of the lights, you will probably want to have your ambient light at a fairly low level (so you get the effect of lights at night or in a dim room) or you can use a fairly high ambient level if you want to see everything in the scene with a reasonably uniform light. If you want to emphasize shapes, use a fairly low ambient light so the shading can bring out the variations in the surface.

Diffuse light comes from specific light sources and is reflected by the surface of the object at a particular wavelength depending on properties of the object's material. The general model for diffuse light, used by OpenGL and other APIs, is based on the idea of brightness, or light energy per unit area. A light emits a certain amount of energy per unit area in the direction it is shining, and when this falls on a surface, the intensity of light seen on the surface is proportional to the surface area illuminated by the unit area of the light. As the diagram in Figure 9.1 shows, one unit of area as seen by the light, or one unit of area perpendicular to the light direction, illluminates an area of $1/\cos(\Theta)$ in the surface. So if we have L_D light energy per unit of area in the light direction, we have $L_D \cos(\Theta)$ units of light energy per unit of area on the surface if the cosine is positive. As the angle of incidence of the light on the surface decreases, the amount of light at the surface becomes reduced, going to zero when the light is parallel to the surface. Because it is impossible to talk about "negative light," we replace any negative value of the cosine with zero, which eliminates diffuse light on surfaces facing away from the light.

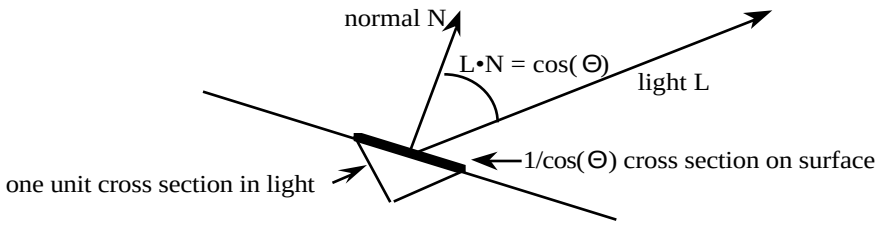


Figure 9.1: diffuse lighting

Now that we know the amount of diffuse light energy per unit of surface, how does that appear to the eye? Diffuse light is reflected from a surface according to Lambert's law that states that the amount of light reflected in a given direction is proportional to the cosine of the angle from the surface in that direction. If the amount of diffuse light per unit surface is D and the unit vector from the surface to your eye is named E, then the amount of energy reflected from one unit of surface to your eye is $D\cos(\Theta) = D\ (E \cdot N)$. But the unit of surface area is not seen by your eye to be a unit of area; its area is $\cos(\Theta) = E \cdot N$. So the intensity of light you perceive at the surface is the ratio of the energy your eye receives and the area your eye sees, and this intensity is simply D—which is the same, no matter where your eye is. Thus the location of the eye point does not participate in the diffuse light computation, which fits our observation things do not change color as the angle between their normals and our eye direction does not change as we move around.

Based on the discussions above, we now have the diffuse lighting calculation that computes the intensity of diffuse light as

$$D = L_D\ \ C_D\ \ \cos(\Theta) = L_D\ \ C_D\ \ (L \cdot N)$$

for the value of the diffuse light L_D from each light source and the ambient property of the material C_D, which shows why we must have surface normals in order to calculate diffuse light. Again, if you have several lights, the diffuse light is $D = \sum L_D\ \ C_D\ \ (L \cdot N)$ where the sum is over all the lights (including a different light vector L for each light). Our use of the dot product instead of the cosine assumes that our normal vector N and light vector L are of unit length, as we discussed when we introduced the dot product. This computation is done separately for each light source and each object, because it depends on the angle from the object to the light.

Diffuse light interacts with the objects it illuminates in a way that produces the color we see in the objects. The object does not reflect all the light that hits it; rather, it absorbs certain wavelengths (or colors) and reflects the others. The color we see in an object, then, comes from the light that is reflected instead of being absorbed, and it is this behavior that we will specify when we define the diffuse property of materials for our scenes.

Specular light is a surface phenomenon that produces bright highlights on shiny surfaces. Specular light depends on the smoothness and electromagnetic properties of the surface, so smooth metallic objects (for example) reflect light well. The energy in specular light is not reflected according to Lambert's law as diffuse light is, but is reflected with the angle of incidence equal to the angle of reflection, as illustrated in Figure 9.2. Such light may have a small amount of "spread" as it leaves the object, depending on the shininess of the object, so the standard model for specular light allows you to define the shininess of an object to control that spread. Shininess is controlled by a parameter called the *specularity coefficient* which gives smaller, brighter highlights as it increases and makes the material seem increasingly shiny, as shown in the three successive figures of Figure 9.3. The computation of the reflection vector was described in Chapter 4, and this reflection vector is calculated as $R = 2(N \cdot L)N - L$ when we take into effect the different direction of the light vector L and use the names in Figure 9.2.

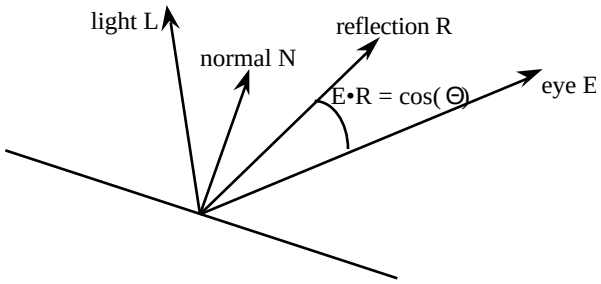


Figure 9.2: specular lighting

The specular light seen on an object in the image is computed by the equations

$$S = L_S \quad C_S \quad \cos^N(\Theta) = L_S \quad C_S \quad (E \cdot R)^N$$

for a light's specularity value L_S and the object's specular coefficient C_S. Again, we assume the eye vector *E* and reflection vector *R* have unit length in order to use the dot product for the cosine. The specular light depends on the angle between the eye and the light reflection, because it is light that is reflected directly from the surface of an object in a mirror-like way. Because it describes the extent to which the surface acts shiny, the specular coefficient is also called the *shininess* coefficient, and larger values give shinier materials. Shininess is a relative term, because even rather shiny materials may have small surface irregularities that keep the reflection from being perfectly mirror-like. Note the visual effect of increasing the specularity coefficient: the highlight gets smaller and more focused—that is, the sphere looks shinier and more polished. Thus the specular or shininess coefficient is part of the definition of a material. This produces a fairly good model of shininess, because for relatively large values of *N* (for example, *N* near or above 30) the function $\cos^N(\Theta)$ has value very near one if the angle Θ is small, and drops off quickly as the angle increases, with the speed of the dropoff increasing as the power is increased. The specular light computation is done separately for each light and each object, because it depends on the angle from the object to the light (as well as the angle to the eye point). This calculation depends fundamentally on both the direction from the object to the light and the direction of the object to the eye, so you should expect to see specular light move as objects, lights, or your eye point moves.

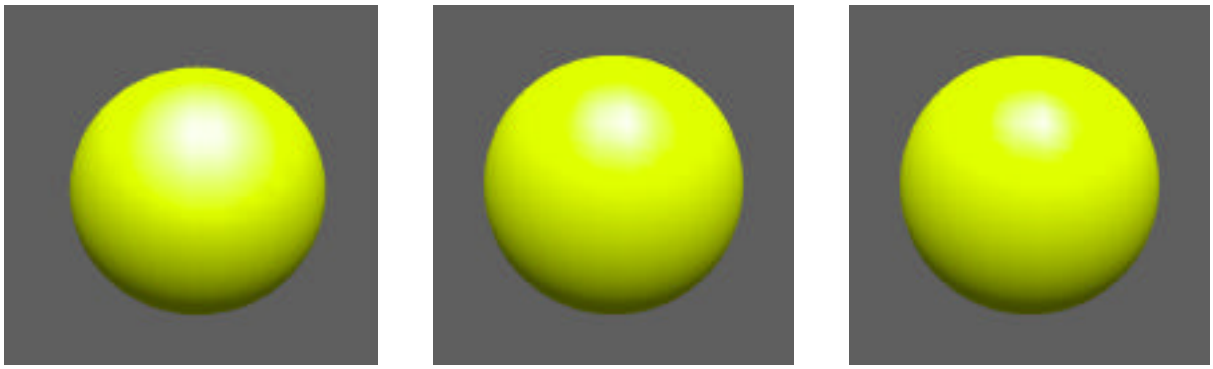


Figure 9.3: specular highlights with specular coefficients 20, 50, and 80 (left, center, and right), respectively

Specular light behaves quite differently from diffuse light in the way it interacts with objects. We generally assume that no light is absorbed in the specular process so that the color of the specular highlight is the same as the color of the light, although it may be possible to specify a material that behaves differently.

Because both diffuse and specular lighting need to have normals to the surface at each vertex, we need to remind ourselves how we get normals to a surface. One way to do this is analytically; we know that the normal to a sphere at any given point is in the direction from the center of the sphere to the point, so we need only know the center and the point to calculate the normal. In other cases, such as surfaces that are the graph of a function, it is possible to calculate the directional derivatives for the function at a point and take the cross product of these derivatives, because the derivatives define the tangent plant to the surface at the point. But sometimes we must calculate the normal from the coordinates of the polygon, and this calculation was described in the discussion of mathematical fundamentals by taking the cross product of two adjacent edges of the polygon in the direction the edges are oriented.

So with the mechanics of computing these three light values in hand, we consider the constants that appeared in the calculations above. The ambient constant is the product of the ambient light component and the ambient material component, each calculated for the red, green, and blue parts respectively. Similarly the diffuse and specular constants are the products of their respective light and material components. Thus a white light and any color of material will produce the color of the material; a red light and a red material will produce a red color; but a red light and a blue material will produce a black color, because there is no blue light to go with the blue material and there is no red material to go with the red light. The final light at any point is the sum of these three parts: the ambient, diffuse, and specular values, each computed for all three RGB components. If any component has a final value larger then one, it is clamped to have value 1.

When you have multiple lights, they are treated additively—the ambient light in the scene is the sum of any overall ambient light for the entire scene plus the ambient lights of the individual lights, the diffuse light in the scene is the sum of the diffuse lights of the individual lights, and the specular light in the scene is the sum of the diffuse lights of the individual lights. As above, if these sums exceed one in any one component, the value is clamped to unity.

Later in this chapter we will discuss shading models, but here we need to note that all our lighting computations assume that we are calculating the light at a single vertex on a model. If we choose to do this calculation at only one point on each polygon, we can only get a single color for the polygon, which leads to the kind of lighting called flat shading. If we wanted to do smooth shading, which can give a much more realistic kind of image, we would need to do a lighting computation could give us a color for each vertex, which would require us to determine a separate normal for each vertex. If the vertex is part of several polygons and we want to calculate a normal for the vertex that we can use for all the polygons, we can calculate a separate normal based on each of the polygons and then average them to get the normal for the vertex. The individual colors for the vertices are then used to calculate colors for all the points in the polygon, as is discussed in more detail when we discuss shading later in this chapter.

Note that none of our light computation handles shadows, however, because shadows depend on the light that reaches the surface, which is a very different question from the way light is reflected from the surface. Shadows are difficult and are handled in most graphics APIs with very specialized programming. We will discuss a simple approach to shadows based on texture mapping later in the book.

Surface normals

As we saw above, you need to calculate normals to the surface in order to compute diffuse and specular light. This is often done by defining normal vectors to the surface in the specifications of the geometry of a scene to allow the lighting computation to be carried out. Processes for computing normals were described in the early chapter on mathematical fundamentals. These can involve analysis of the nature of the object, so you can sometimes compute exact normals (for example, if you are displaying a sphere, the normal at any point has the same direction as the

radius vector). If an analytic calculation is not available, normals to a polygonal face of an object can be computed by calculating cross products of the edges of the polygon. However, it is not enough merely to specify a normal; you need to have unit normals, normal vectors that are exactly one unit long (usually called normalized vectors). It can be awkward to scale the normals yourself, and doing this when you define your geometry may not even be enough because scaling or other computations can change the length of the normals. In many cases, your graphics API may provide a way to define that all normals are to be normalized before they are used.

In a previous chapter, we saw that the cross product of two vectors is another vector that is perpendicular to both of them. If we have a polygon (always considered here to be convex), we can calculate a vector parallel to any edge by calculating the difference of the endpoints of the edge. If we do this for two adjacent edges, we get two vectors that lie in the polygon's plane, so their cross product is a vector perpendicular to both edges, and hence perpendicular to the plane, and we have created a normal to the polygon. When we do this, it is a very good idea to calculate the length of this normal and divide this length into each component of the vector in order to get a unit vector because most APIs' lighting model assumes that our normals have this property.

It is also possible to compute normals from analytic considerations if we have the appropriate modeling to work with. If we have an object whose surface is defined by differentiable parametric equations, then we can take appropriate derivatives of those equations to get two directions for the tangent plane to the surface, and take their cross product to get the normal.

In both cases, we say we have the "normal" but in fact, a surface normal vector could go either way from the surface, either "up" or "down". We need to treat surface normals consistently in order to get consistent results for our lighting. For the approach that takes the cross product of edges, you get this consistency by using a consistent orientation for the vertices of a polygon, and this is usually done by treating the vertices, and thus the edges, as proceeding in a counterclockwise direction. Another way to say this is to take any interior point and the line from the interior point to the first vertex. Then the angle from that line to the line from the point to any other vertex will increase as the vertex index increases. The orientation for analytic normals is simpler. If we have the x- and *y*-direction vectors in the tangent plane to the surface, we recall the right-hand rule for cross products and always take the order (*x*-direction)x(*y*-direction) in order to get the *z*-direction, which is the normal.

Because a polygon is a plane figure, it has two sides which we will call the *front* side and *back* side. If we take the conventions above for the normal, then the normal will always point out of a polyhedron, will always point up on a surface defined as a function of two variables, or will always point toward the side of a polygon in which the vertices appear to be in counterclockwise order. This side towards which the normal points is called the front side, and this distinction is important in material considerations below as well as other graphical computations.

Light properties

As we noted at the beginning of this chapter, we will focus on a local illumination model for our discussion of lighting. Because all the lighting comes from lights you provide, lights are critical components of your modeling work in defining an image. In the chapter on the principles of modeling, we saw how to include lights in your scene graph for an image. Along with the location of each light, which is directly supported by the scene graph, you will want to define other aspects of the light, and these are discussed in this section.

Your graphics API allows you to define a number of properties for a light. Typically, these can include its position or its direction, its color, how it is attenuated (diminished) over distance, and whether it is an omnidirectional light or a spotlight. We will cover these properties lightly here but will not go into depth on them all, but the properties of position and color are critical. The other

properties are primarily useful is you are trying to achieve a particular kind of effect in your scene. The position and color properties are illustrated in the example at the end of this chapter.

Positional lights

When we want a light that works as if it were located within your scene, you will want your light to have an actual position in the scene. To define a light that has position, you will set the position as a four-tuple of values whose fourth component is non-zero (typically, you will set this to be 1.0). The first three values are then the position of the light and all lighting calculations are done with the light direction from an object set to the vector from the light position to the object.

Spotlights

Unless you specify otherwise, a positional light will shine in all directions. If you want a light that shines only in a specific direction, you can define the light to be a spotlight that has not only a position, but also other properties such as a direction, a cutoff, and a dropoff exponent, as you will see from the basic model for a spotlight shown in Figure 9.4. The direction is simply a 3D vector that is taken to be parallel to the light direction, the cutoff is assumed to be a value between 0.0 and 90.0 that represents half the spread of the spotlight and determines whether the light is focused tightly or spread broadly (a smaller cutoff represents a more focused light), and the dropoff exponent controls how much the intensity drops off between the centerline of the spotlight and the intensity at the edge.

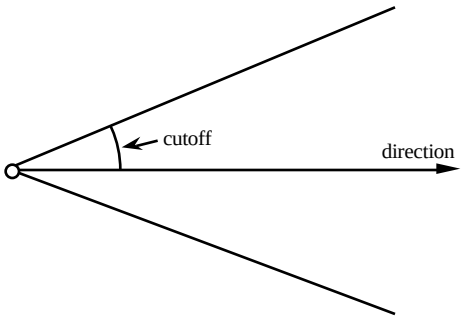


Figure 9.4: spotlight direction and cutoff

In more detail, if a spotlight has position P, dropoff d, direction D, and cutoff Θ, the light energy at a point Q in a direction V from the spotlight would then be zero if the absolute value of the dot product $(Q–P)\bullet D$ were larger than $\cos(\Theta)$, or would be multiplied by $((Q–P)\bullet D)^d$ if the absolute value of the dot product is less than $\cos(\Theta)$.

Attenuation

The physics of light tells us that the energy from a light source on a unit surface diminishes as the square of the distance from the light source from the surface. This diminishing is called attenuation, and computer graphics can model that behavior in a number of ways. An accurate model would deal with the way energy diffuses as light spreads out from a source which would lead to a light that diminishes as the square of the distance d from the light, multiplying the light enerty by k/d^2, and the graphics system would diminish the intensity of the light accordingly. However, the human perceptual system is more nearly logarithmic than linear in the way we see light, so we do not recognize this kind of diminishing light as realistic, and we probably would

need to use an attenuation that drops off more slowly. Your graphics API will probably give you some options in modeling attenuation.

Directional lights

Up to now, we have talked about lights as being in the scene at a specific position. When such lights are used, the lighting model takes the light direction at any point as the direction from the light position to that point. However, if we were looking for an effect like sunlight, we want light that comes from the same direction at all points in the scene. In effect, we want to have a light at a point at infinity. If your graphics API supports directional lights, there will be a way to specify that the light is directional instead of positional and that simply defines the direction from which the light will be received.

Positioning and moving lights

Positional lights can be critical components of a scene, because they determine how shapes and contours can be seen. As we noted in the chapter on modeling, lights are simply another part of the model of your scene and affected by all the transformations present in the modelview matrix when the light position is defined. A summary of the concepts from the scene graph will help remind us of the issues here.

- If the light is to be at a fixed place in the scene, then it is at the top level of the scene graph and you can define its position immediately after you set the eye point. This will create a position for the light that is independent of the eye position or of any other modeling in the scene.
- If the light is to be at a fixed place relative to the eye point, then you need to define the light position and other properties before you define the eye position. The light position and properties are then modified by the transformations that set the eye point, but not by any subsequent modeling transformations in the scene.
- If the light is to be at a fixed place relative to an object in the scene, then you define the light position as a branch of the group node in the scene graph that defines the object. Anything that affects the object will then be done above that group node, and will affect the light in the same way as it does the rest of the object.
- If the light is to move around in the scene on its own, then the light is a content node of the scene graph and the various properties of the light are defined as that node is set.

The summary of this modeling is that a positional light is treated simply as another part of the modeling process and is managed in the same way as any other object would be.

Materials

Lighting involves both the specification of the lights in a scene and the light-related properties of the objects in the scene. If you want to use lighting in creating a scene, you must specify both of these: the properties of your lights, and the properties of the materials of your objects. In this section we discuss the nature of material specifications. Implementing lighting for a scene involves putting these all together as is discussed in the example at the end of this chapter.

As we saw above, each object participates in determining the reflected light that makes up its color when it is displayed. In the discussion of the three components of light, we saw four material properties C_A, the reflectivity of the material in ambient light; C_D the reflectivity of the material in diffuse light; C_S, the reflectivity of the material in specular light; and N, the shininess coefficient. The three reflectivity terms involve color, so they are specified in RGB terms, while the shininess coefficient is the exponent of the dot produce $R \bullet E$ in the specular component. These all take part in the computations of the color of the object from the lighting model, and are called the material specifications. They need to be defined for each object in your scene in order to allow the lighting calculations to be carried out. Your graphics API will allow you to see these as part of your

modeling work; they should be considered as part of the appearance information you would include in a shape node in your scene graph. The API may have some other kinds of behavior for materials, such as specifications of the front and back sides of a material separately; this is the case for OpenGL, but it may vary between APIs.

All the discussion of lighting above assumed that an object is reflective, but an object can also be *emissive*, that is, send out light of its own. Such a light simply adds to the light of the object but does not add extra light to the scene, allowing you to define a bright spot to present something like an actual light in the scene. This is managed by defining a material to have an emissive light property, and the final lighting calculations for this material adds the components of the light emission to the other lighting components when the object's color is computed.

Shading

Shading is the process of computing the color for the components of a scene. It is usually done by calculating the effect of light on each object in a scene to create an effective lighted presentation. The shading process is thus based on the physics of light, and the most detailed kinds of shading computation can involve deep subtleties of the behavior of light, including the way light scatters from various kinds of materials with various details of surface treatments. Considerable research has been done in those areas and any genuinely realistic rendering must take a number of surface details into account.

Most graphics APIs do not have the capability to do these detailed kinds of computation. The usual beginning API such as OpenGL supports two shading models for polygons: flat shading and smooth shading. You may choose either, but smooth shading is usually more pleasing and can be somewhat more realistic. Unless there is a sound reason to use flat shading in order to represent data or other communication concepts more accurately, you will probably want to use smooth shading for your images. We will briefly discuss just a bit more sophisticated kinds of shading, even though the beginning API cannot directly support them.

Definitions

Flat shading of a polygon presents each polygon with a single color. This effect is computed by assuming that each polygon is strictly planar and all the points on the polygon have exactly the same kind of lighting treatment. The term flat can be taken to mean that the color is flat (does not vary) across the polygon, or that the polygon is colored as though it is flat (planar) and thus does not change color as it is lighted. This is the effect you will get if you simply set a color for the polygon and do not use a lighting model (the color is flat), or if you use lighting and materials models and then display the polygon with a single normal vector (the polygon is flat). This single normal allows you only a single lighting computation for the entire polygon, so the polygon is presented with only one color.

Smooth shading of a polygon displays the pixels in the polygon with smoothly-changing colors across the surface of the polygon. This requires that you provide information that defines a separate color for each vertex of your polygon, because the smooth color change is computed by interpolating the vertex colors across the interior of the triangle with the standard kind of interpolation we saw in the graphics pipeline discussion. The interpolation is done in screen space after the vertices' position has been set by the projection, so the purely linear calculations can easily be done in graphics cards. This per-vertex color can be provided by your model directly, but it is often produced by per-vertex lighting computations. In order to compute the color for each vertex separately you must define a separate normal vector for each vertex of the polygon so that the lighting model will produce different colors at each vertex.

Every graphics API will support flat shading in a consistent way, but different graphics APIs may treat smooth shading somewhat differently, so it is important for you to understand how your particular API handles this. The simplest smooth shading is done by calculating color at the vertices of each polygon and then interpolating the colors smoothly across the polygon. If the polygon is a triangle, you may recall that every point in the triangle is a convex combination of the vertices, so you may simply use that same convex combination of the vertex colors. As computer graphics becomes more sophisticated, however, we will see more complex kinds of polygon shading in graphics APIs so that the determination of colors for the pixels in a polygon will become increasingly flexible.

Examples of flat and smooth shading

We have seen many examples of polygons earlier in these notes, but we have not been careful to distinguish between whether they were presented with flat and smooth shading. Figure 9.5 shows see two different images of the same relatively coarsely-defined function surface with flat shading (left) and with smooth shading (right), to illustrate the difference. Clearly the smooth-shaded image is much cleaner, but there are still some areas where the triangles change direction very quickly and the boundaries between the triangles still show as color variations in the smoothly-shaded image. Smooth shading is very nice—probably nicer than flat shading in many applications—but it isn't perfect.

The computation for this smooth shading uses simple polygon interpolation in screen space. Because each vertex has its own normal, the lighting model computes a different color for each vertex. The interpolation then calculates colors for each pixel in the polygon that vary smoothly across the polygon interior, providing a smooth color graduation across the polygon. This interpolation is called *Gouraud shading* and is one of the standard techniques for creating images. It is quick to compute but because it only depends on colors at the polygon vertices, it can miss lighting effects within polygons. Visually, it is susceptible to showing the color of a vertex more strongly along an edge of a polygon than a genuinely smooth shading would suggest, as you can see in the right-hand image in Figure 9.5. Other kinds of interpolation are possible that do not show some of these problems, though they are not often provided by a graphics API, and one of these is discussed below.

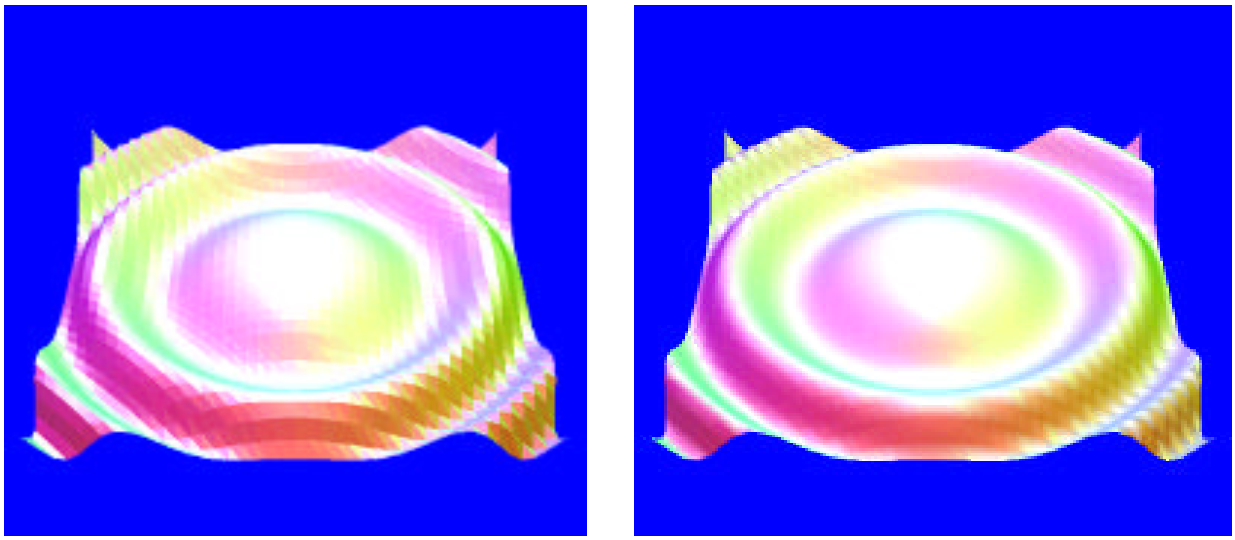


Figure 9.5: a surface with flat shading (left) and the same surface with smooth shading (right)

An interesting experiment to help you understand the properties of shaded surfaces is to consider the relationship between smooth shading and the resolution of the display grid. In principle, you should be able to use fairly fine grid with flat shading or a much coarser grid with smooth shading to achieve similar results. You should define a particular grid size and flat shading, and try to find the smaller grid that would give a similar image with smooth shading. Figure 9.6 is an example of this experiment. The surface still shows a small amount of the faceting of flat shading but avoids much of the problem with quickly-varying surface directions of a coarse smooth shading. It is probably superior in many ways to the smooth-shaded polygon of Figure 9.5. It may be either faster or slower than the original smooth shading, depending on the efficiency of the polygon interpolation in the graphics pipeline. This is an example of a very useful experimental approach to computer graphics: if you have several different ways to approximate an effect, it can be very useful to try all of them that make sense and see which works better, both for effect and for speed, in a particular application!

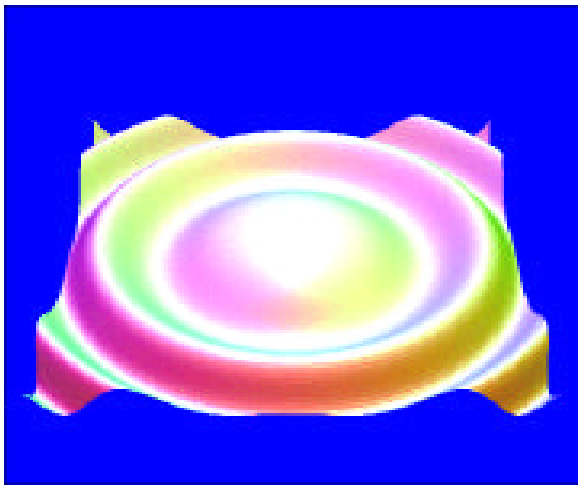


Figure 9.6: a flat-shaded image with three times as many subdivisions in each direction as the previous figure

Calculating per-vertex normals

The difference between the programming for these two parts of Figure 10.1 is that the flat-shaded model uses only one normal per polygon (calculated by computing the cross product of two edges of each triangle or by computing an analytic normal for any point in the polygon), while the smooth-shaded model uses a separate normal per vertex (in this case, calculated by using analytic processes to determine the exact value of the normal at the vertex). It can take a bit more work to compute the normal at each vertex instead of only once per polygon, but that is the price one must pay for smooth shading.

There are a number of ways you may calculate the normal for a particular vertex of a model. You may use an interpolation technique, in which you compute a weighted average of the normals of each of the polygons that includes the vertex, or you may use an analytic computation. The choice of technique will depend on the information you have for your model.

Averaging polygon normals

In the interpolation technique, you can calculate the normal N at a vertex by computing the weighted average of the normals for all the polygons that meet at the vertex as

$$N = (\sum a_i N_i)/(\sum a_i),$$

with the sum taken over all indices *i* of polygons P_i that include this vertex, where each polygon P_i has a normal N_i and has angle a_i at the vertex in question. Each angle a_i can be calculated easily as the inverse cosine of the dot product of the two edges of the polygon P_i that meet at the vertex.

Analytic computations

In the example of Figure 9.5, an analytic approach to computing the normal *N* at each vertex was possible as described in the previous chapter because the surface was defined by $0.3 \cos(x \quad x + y \quad y + t)$, a clean, closed-form equation. In the smooth-shaded example, we were able to calculate the vertex normals by using the analytic directional derivatives at each vertex: the derivatives in the x direction and y direction respectively are

$\partial f / \partial x = -0.6 \quad x \quad \sin(x \quad x + y \quad y + t)$ and

$\partial f / \partial y = -0.6 \quad y \quad \sin(x \quad x + y \quad y + t)$.

These are used to calculate the tangent vectors in these directions, and the cross products of those tangent vectors were computed to get the vertex normal. This is shown in the code sample at the end of this chapter. It can also be possible to get exact normals from other kinds of models; we saw in an early chapter in these notes that the normals to a sphere are simply the radius vectors for the sphere, so a purely geometric model may also have exactly-defined normals. In general, when models permit you to carry out analytic or geometric calculations for normals, these will be more exact and will give you better results than using an interpolation technique.

Other shading models

You cannot and must not assume that the smooth shading model of a simply API such as OpenGL is an accurate representation of smooth surfaces. It assumes that the surface of the polygon varies uniformly, it only includes per-vertex information in calculating colors across the polygon, and it relies on a linear behavior of the RGB color space that is not accurate, as you saw when we talked about colors. Like many of the features of any computer graphics system, it approximates a reality, but there are better ways to achieve the effect of smooth surfaces. For example, there is a shading model called *Phong shading* that requires the computation of one normal per vertex and uses the interpolated values of the normals themselves to compute the color at each pixel in the polygon, instead of simply interpolating the vertex colors. Interpolating the normals is much more complicated than interpolating colors, because the uniformly-spaced pixels in screen space do not come from uniformly-spaced points in 3D eye space or 3D model space; the perspective projection involves a division by the *Z*-coordinate of the point in eye space. This makes normal interpolation more complex—and much slower—than color interpolation and takes it out of the range of simple graphics APIs. However, the Phong shading model behaves like a genuinely smooth surface across the polygon, including picking up specular highlights within the polygon and behaving smoothly along the edges of the polygon. The details of how Gouraud and Phong shading operate are discussed in any graphics textbook. We encourage you to read them as an excellent example of the use of interpolation as a basis for many computer graphics processes.

The Phong shading model assumes that normals change smoothly across the polygon, but another shading model is based on controlling the normals across the polygon. Like the texture map that we describe later and that creates effects that change across a surface and are independent of the colors at each vertex, we may create a mapping that alters the normals in the polygon so the shading model can create the effect of a bumpy surface. This is called a *bump map*, and like Phong shading the normal for each individual pixel is computed separately. Here the pixel normal is computed as the normal from Phong shading plus a normal computed from the bump map by the gradient of the color. The color of each individual pixel is then computed from the standard lighting model. Figure 9.7 shows an example of the effect of a particular bump map. Note that the bump map itself is defined simply a 2D image where the height of each point is defined by the

color; this is called a height field. Elevations or distances are sometimes presented in this way, for example in terrain modeling.

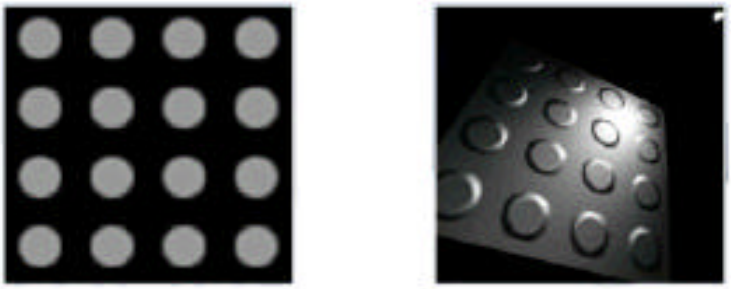


Figure 9.7: a bump map defined as a height field, left, and the bump map applied to a specular surface

The shading models that have been described so far are all based on the simple lighting model of the previous chapter, which assumes that light behavior is uniform in all directions from the surface normal (that is, the lighting is *isotropic*). However, there are some materials where the lighting parameters differ depending on the angle around the normal. Such materials include brushed metals and the surface of a CD, for example, and the shading for these materials is called *anisotropic*. Here the simple role of the angle from the normal of the diffuse reflection, and the angle from the reflected light in the specular reflection, are replaced by a more complex function called the *bidirectional reflection distribution function* (or *BRDF*) that depends typically on both the latitude Θ and longitude Φ angle of the eye and of the light from the point being lighted: $\rho(\Theta_e, \Phi_e, \Theta_l, \Phi_l)$. The BRDF may also take into account behaviors that differ for different wavelengths of light. The lighting calculations for such materials, then, may involve much more complex kinds of computation than the standard isotropic model and are beyond the scope of simple graphics APIs, but you will find this kind of shading in some professional graphics tools. Figure 9.8 shows the effect on a red sphere of applying flat, smooth, and Phong shading, and an anisotropic shading.

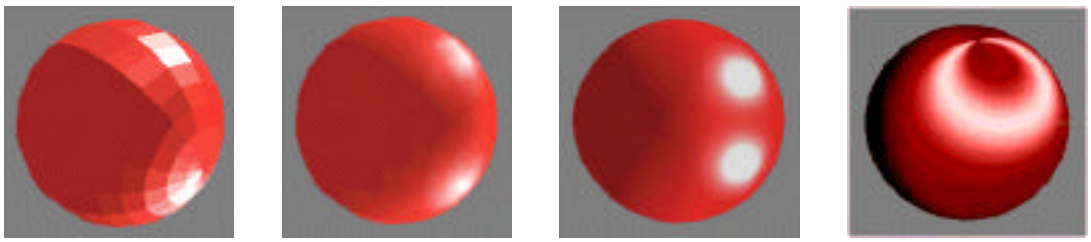


Figure 9.8: a sphere presented with flat shading (left), smooth shading (second), Phong shading (third) and an anisotropic shading (right).

Note that the smooth-shaded sphere shows some facet edges and the specular reflection is not quite smoothly distributed over the surface, while the facet edges and the specular reflection in the Phong shaded sphere are quite a bit smoother and less broken up.

Vertex and pixel shaders

One of the recent advances in shading that is somewhat ahead of most graphics APIs is providing a shading language with which the programmer can define shading at a much more detailed level than we have discussed so far. By allowing you to define a programmatic way to shade both

vertices and individual pixels, it is possible to develop anisotropic shaders, motion blur, bump mapping, and many other very sophisticated kinds of effects. There is ongoing research in making this accessible to the programmer through vertex shading languages that could be part of a graphics API, and in the last chapter we saw how they can fit into the structure of the rendering pipeline. We look forward to a great deal of development in this direction in the near future.

Global illumination

There is another approach to lighting that takes a more realistic view than the local illumination model. In any actual scene, the light that does not come from direct lights is not simply an ambient value; it is reflected from every object, every surface in the scene. Lighting that accounts for this uses a *global illumination model*, so-called because light is computed globally for the entire scene independent of the viewpoint instead of being computed for each polygon separately in a way that depends on the viewpoint, as we did earlier in this chapter.

Global illumination models include *radiosity*, which assumes that every surface can emit light and calculates the emission in a sequence of steps that converges to an eventual stable light. In the first step, light sources emit their light and any other source is zero. The light energy arriving at each surface is calculated and stored with the surface, and in the next step, that energy is emitted according to properties of the material. This step is repeated until the difference between energies at each point from one step to the next is small, and one assumes that the emission from each point is then known. When the scene is displayed, each point is given a color consistent with the energy emitted at that point. In practice, the scene is not handled as a set of points, however; every object in the scene is subdivided into a number of patches having a simple geometry, and the calculations are done on a per-patch basis.

Another global illumination approach is called *photon mapping*. In this approach the light energy is modeled by emitting photons from each light source and tracing them as they hit various surfaces in the scene. As a photon hits a surface it is accumulated to the surface and, may be emitted from the surface based on the properties of the surface and on a probability calculation. This is done until a sufficient (and surprisingly small) number of photons have been traced through the scene, and then the overall illumination is derived from the accumulated results. You can see that this is quite similar in some ways to the radiosity concept but the computation is quite different.

Because of the way these global illumination processes work, there is no question of shading models with them. Any unique kinds of shading comes out in the way the light energy arriving at a surface is passed on to the next surface.

One of the advantages of global illumination is that once you have computed the light at each point in the scene, displaying the scene can be done very rapidly. This is a good example of an asymmetric process, because you can put in a large amount of processing to create the lighting for the model, but once you have done so, you need not do much processing to display the model. This has a number of interesting and useful applications but is not yet supported by basic graphics APIs—though it may be in the future.

Local illumination

In contrast to global illumination models, where the energy reflected from every surface is taken into account, local illumination models assume that light energy comes only from light sources. This is the approach taken by OpenGL, where the light at any point is only that accounted for by the ambient, diffuse, and specular light components described earlier in this chapter. This is handled by defining the properties of lights and of materials relative to these three components, as we describe in this section.

Lights and materials in OpenGL

Several times above we suggested that a graphics API would have facilities to support several of the lighting issues we discussed. Here we will outline the OpenGL support for lighting and materials so you can use these capabilities in your work. In some of these we will use the form of the function that takes separate *R*, *G*, and *B* parameters (or separate *X*, *Y*, and *Z* coordinates), such as `glLightf(light, pname, set_of_values)`, while in others we will use the vector form that takes 3-dimensional vectors for colors and points, but in come cases we will use the vector form such as `glLightfv(light, pname, vector_values)`, and you may use whichever form fits your particular design and code best.

As is often the case in OpenGL, there are particular names that you must use for some of these values. Lights must be named `GL_LIGHT0` through `GL_LIGHT7` for standard OpenGL (some implementations may allow more lights, but eight possible lights is standard), and the parameter name `pname` must be one of the available light parameters

`GL_AMBIENT`,
`GL_DIFFUSE`,
`GL_SPECULAR`,
`GL_POSITION`,
`GL_SPOT_DIRECTION`,
`GL_SPOT_EXPONENT`,
`GL_SPOT_CUTOFF`,
`GL_CONSTANT_ATTENUATION`,
`GL_LINEAR_ATTENUATION`, or
`GL_QUADRATIC_ATTENUATION`

In this section we will discuss the properties of OpenGL lights that lead to these parameters.

Specifying and defining lights

When you begin to plan your scene and are designing your lighting, you may need to define your light model with the `glLightModel(...)` function. This will allow you to define some fundamental properties of your lighting. Perhaps the most important use of this function is defining is whether your scene will use one-sided or two-sided lighting, which is chosen with the function

`glLightModel[f|i](GL_LIGHT_MODEL_TWO_SIDE, value)`.

where `[f|i]` means that you use either the letter `f` or the letter `i` to indicate whether the parameter value is real or integer. If the (real or integer) value of the numeric parameter is 0, one-sided lighting is used and only the front side of your material is lighted; if the value is non-zero, both front and back sides of your material are lighted. Other uses of the function include setting a global ambient light, discussed below, and choosing whether specular calculations are done by assuming the view direction is parallel to the *Z*-axis or the view direction is towards the eye point This is determined by the function

`glLightModel[f|i](GL_LIGHT_MODEL_LOCAL_VIEWER,value)`,

with a value of 0 meaning that the view direction is parallel to the *Z*-axis and non-zero that the view direction is toward the origin. The default value is 0.

OpenGL allows you to define up to eight lights for any scene. These lights have the symbolic names `GL_LIGHT0` … `GL_LIGHT7`, and you create them by defining their properties with the `glLight*(...)` functions before they are available for use. You define the position and color of your lights (including their ambient, specular, and diffuse contributions) as illustrated for the light

GL_LIGHT0 by the following position definition and definition of the first of the three lights in the three-light example

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
glLightfv(GL_LIGHT0, GL_AMBIENT,  amb_color0 );
glLightfv(GL_LIGHT0, GL_DIFFUSE,  diff_col0 );
glLightfv(GL_LIGHT0, GL_SPECULAR, spec_col0 );
```

Here we use a light position and specific light colors for the specular, diffuse, and ambient colors that we must define in separate statements such as those below.

```
GLfloat light_pos0 = { ..., ..., ... };
GLfloat diff_col0  = { ..., ..., ... };
```

In principle, both of these vectors are four-dimensional, with the fourth value in the position vector being a homogeneous coordinate value and with the fourth value of the color vector being the alpha value for the light. We have not used homogeneous coordinates to describe our modeling, but they are not critical for us. We have used alpha values for colors, of course, but the default value for alpha in a color is 1.0 and unless you want your light to interact with your blending design somehow, we suggest that you use that value for the alpha component of light colors, which you can do by simply using RGB-only light definitions as we do in the example at the end of this chapter.

As we noted earlier in this chapter, you must define normals to your objects' surfaces for lighting to work successfully. Because the lighting calculations involve cosines that are calculated with dot products with the normal vector, however, you must make sure that your normal vectors are all of unit length. You can ensure that this is the case by enabling automatic normalization with the function call glEnable(GL_NORMALIZE) before any geometry is specified in your display function.

Before any light is available to your scene, the overall lighting operation must be enabled and then each of the individual lights to be used must also be enabled. This is an easy process in OpenGL. First, you must specify that you will be using lighting models by invoking the standard enable function

```
glEnable(GL_LIGHTING);   // so lighting models are used
```

Then you must identify the lights you will be using by invoking an enable function for each light, as illustrated by the following setup of all three lights for the three-light case of the example below:

```
glEnable(GL_LIGHT0);     // use LIGHT0
glEnable(GL_LIGHT1);     // and LIGHT1
glEnable(GL_LIGHT2);     // and LIGHT2
```

Lights may also be disabled with the glDisable(...) function, so you may choose when to have a particular light active and when to have it inactive in an animation or when carrying out a particular display that may be chosen, say, by a user interaction.

In addition to the ambient light that is contributed to your scene from each of the individual lights' ambient components, you may define an overall ambient light for the scene that is independent of any particular light. This is done with the function:

```
glLightModelf(GL_LIGHT_MODEL_AMBIENT, r, g, b, a)
```

and the value of this light is added into the overall ambient lighting computation.

The remaining properties of lights that we discussed earlier in this chapter are also straightforward to set in OpenGL. If you want a particular light to be a spotlight, you will need to set the direction, cutoff, and dropoff properties that we described earlier in this chapter, as well as the standard position property. These additional properties are set with the glLightf*(...) functions as follows:

```
glLightf(light, GL_SPOT_DIRECTION, -1.0, -1.0, -1.0);
glLightf(light, GL_SPOT_CUTOFF, 30.0);
glLightf(light, GL_SPOT_EXPONENT, 2.0);
```

If you do not specify the spotlight cutoff and exponent, these are 180° (which means that the light really isn't a spotlight at all) and the exponent is 0. If you do set the spotlight cutoff, the value is limited to lie between 0 and 90, as we described earlier.

Attenuation is not modeled realistically by OpenGL, but is set up in a way that can make it useful. There are three components to attenuation: constant, linear, and quadratic. The value of each is set separately as noted above with the symbolic constants `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. If these three attenuation coefficients are A_C, A_L, and A_Q, respectively, and the distance of the light from the surface is `D`, then the light value is multiplied by the attenuation factor

$$A = 1/(A_C + A_L \quad D + A_Q \quad D^2)$$

where D is the distance between the light position and the vertex where the light is calculated. The default values for A_C, A_L, and A_Q (think of constant, linear, and quadratic attenuation terms) are 1.0, 0.0, and 0.0 respectively. The actual values of the attenuation constants can be set by the functions

```
glLightf(GL_*_ATENUATION, value)
```

functions, where the wildcard * is to be replaced by one of the three symbolic constants identified above.

A directional light is specified by setting the fourth component in its position to be zero. The direction of the light is set by the first three components, and these are transformed by the modelview matrix. Such lights cannot have any attenuation properties but otherwise work just like any other light: its direction is used in any diffuse and specular light computations but no distance is ever calculated. An example of the way a directional light is defined would be

```
glLightf(light, GL_POSITION, 10.0, 10.0, 10.0, 0.);
```

Defining materials

In order for OpenGL to model the way a light interacts with an object, the object must be defined in terms of the way it handles ambient, diffuse, and specular light. This means that you must define the color of the object in ambient light and the color in diffuse light. (No, we can't think of any cases where these would be different, but we can't rule out the possibility that this might be used somehow.) You do not define the color of the object in specular light, because specular light is the color of the light instead of the color of the object, but you must define the way the material handles the specular light, which really means how shiny the object is and what color the shininess will be. All these definitions are handled by the `GL_MATERIAL*` function.

OpenGL takes advantage of the two-sided nature of polygons that we described earlier in this chapter, and allows you to specify that for your material you are lighting the front side of each polygon, the back side of each polygon (refer to the earlier discussion for the concept of front and back sides), or both the front and back sides. You do this by specifying your materials with the parameters `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. If you use two-sided lighting, when you specify the properties for your material, you must specify them for both the front side and the back side of the material. You can choose to make these properties the same by defining your material with the parameter `GL_FRONT_AND_BACK` instead of defining `GL_FRONT` and `GL_BACK` separately. This will allow you to use separate colors for the front side and back side of an object, for example, and make it clear which side is being seen in case the object is not closed.

To allow us to define an object's material properties we have the `glMaterial*(...)` function family. These functions have the general form

```
glMaterial[i|f][v](face, parametername, value)
```

and can take either integer or real parameter values ([i|f]) in either individual or vector ([v]) form. The parameter face is a symbolic name that must be one of GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. The value of parametername is a symbolic name whose values can include GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS, or GL_AMBIENT_AND_DIFFUSE. Finally, the value parameter is either a single number, a set of numbers, or a vector that sets the value the symbolic parameter is to have in the OpenGL system. Below is a short example of setting these values, taken from the example at the end of the chapter.

```
GLfloat shininess[]={ 50.0 };
GLfloat white[] = { 1.0, 1.0, 1.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT,   white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SPECULAR,  white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess );
```

This gives the material a very neutral property that can pick up whatever colors the light should provide for its display.

Most of the parameters and values are familiar from the earlier discussion of the different aspects of the lighting model, but the GL_AMBIENT_AND_DIFFUSE parameter is worth pointing out because it is very common to assume that a material has the same properties in both ambient and diffuse light. (Recall that in both cases, the light energy is absorbed by the material and is then re-radiated with the color of the material itself.) This parameter allows you to define both properties to be the same, which supports this assumption.

Setting up a scene to use lighting

To define a triangle with vertex points P[0], P[1], and P[2], compute its normal, and use the calculated normal, we would see code something like this:

```
glBegin(GL_POLYGON);
     // calculate the normal Norm to the triangle
     calcTriangleNorm(p[0],P[1],P[2],Norm);
     glNormal3fv(Norm);
     glVertex3fv(P[0]);
     glVertex3fv(P[1]);
     glVertex3fv(P[2]);
glEnd();
```

Using GLU quadric objects

As we discussed when we introduced the GLU quadric objects in the modeling chapter, the OpenGL system can generate automatic normal vectors for these objects. This is done with the function gluQuadricNormals(GLUquadric* quad, GLenum normal) that allows you to set normal to either GLU_FLAT or GLU_SMOOTH, depending on the shading model you want to use for the object.

An example: lights of all three primary colors applied to a white surface

Some lighting situations are easy to see. When you put a white light on a colored surface, you see the color of the surface, because the white light contains all the light components and the surface has the color it reflects among them. Similarly, if you shine a colored light on a white surface, you see the color of the light because only that color is available. When you use a colored light on a colored surface, however, it gets much more complex because a surface can only reflect colors that come to it. So if you shine a (pure) red light on a (pure) green surface you get no reflection at all, and the surface seems black. You don't see this in the real world because you don't see lights of pure colors, but it can readily happen in a synthetic scene.

Considering the effect of shining colored lights on a white surface, let's look at an example. A white surface will reflect all the light that it gets, so if it gets only a red light, it should be able to reflect only red. So if we take a simple shape (say, a cube) in a space with three colored lights (that are red, green, and blue, naturally), we should see it reflect these different colors. In the example we discuss below, we define three lights that shine from three different directions on a white cube. If you add code that lets you rotate the cube around to expose each face to one or more of the three lights, you will be able to see all the lights on various faces and to experiment with the reflection properties they have. This may let you see the effect of having two or three lights on one of the faces, as well as seeing a single light. You may also want to move the lights around and re-compile the code to achieve other lighting effects.

There is a significant difference between the cube used in this example and the cube used in the simple lighting example in a previous module. This cube includes not only the vertices of its faces but also information on the normals to each face. (A normal is a vector perpendicular to a surface; we are careful to make all surface normals point away from the object the surface belongs to.) This normal is used for many parts of the lighting computations. For example, it can be used to determine whether you're looking at a front or back face, and it is used in the formulas earlier in the chapter to compute both the diffuse light and the specular light for a polygon.

Code for the example

Defining the light colors and positions in the initialization function:

```
GLfloat light_pos0[]={ 0.0, 10.0, 2.0, 1.0 }; //up y-axis
GLfloat light_col0[]={ 1.0,  0.0, 0.0, 1.0 }; //light is red
GLfloat amb_color0[]={ 0.3,  0.0, 0.0, 1.0 };

GLfloat light_pos1[]={ 5.0, -5.0, 2.0, 1.0 }; //lower right
GLfloat light_col1[]={ 0.0,  1.0, 0.0, 1.0 }; //light is green
GLfloat amb_color1[]={ 0.0,  0.3, 0.0, 1.0 };

GLfloat light_pos2[]={-5.0,  5.0, 2.0, 1.0 }; //lower left
GLfloat light_col2[]={ 0.0,  0.0, 1.0, 1.0 }; //light is blue
GLfloat amb_color2[]={ 0.0,  0.0, 0.3, 1.0 };
```

Defining the light properties and the lighting model in the initialization function:

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_color0 );
glLightfv(GL_LIGHT0, GL_SPECULAR, light_col0 );
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_col0 );

glLightfv(GL_LIGHT1, GL_POSITION, light_pos1 ); // light 1
glLightfv(GL_LIGHT1, GL_AMBIENT, amb_color1 );
glLightfv(GL_LIGHT1, GL_SPECULAR, light_col1 );
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_col1 );

glLightfv(GL_LIGHT2, GL_POSITION, light_pos2 ); // light 2
glLightfv(GL_LIGHT2, GL_AMBIENT, amb_color2 );
glLightfv(GL_LIGHT2, GL_SPECULAR, light_col2 );
glLightfv(GL_LIGHT2, GL_DIFFUSE, light_col2 );

glLightModeliv(GL_LIGHT_MODEL_TWO_SIDE, &i ); // two-sided
```

Enabling the lights in the initialization function:

```
glEnable(GL_LIGHTING);   // so lighting models are used
glEnable(GL_LIGHT0);     // we'll use LIGHT0
```

```
glEnable(GL_LIGHT1);     // ...   and LIGHT1
glEnable(GL_LIGHT2);     // ...   and LIGHT2
```

Defining the material color in the function that draws the surface: we must define the ambient and diffuse parts of the object's material specification, as shown below; note that the shininess value must be an array. Recall that higher values of shininess will create more focused and smaller specular highlights on the object. That this example doesn't specify the properties of the material's back side because the object is closed and all the back side of the material is invisible.

```
GLfloat shininess[]={ 50.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT,   white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess );
```

Figure 9.9 shows the cube when it is rotated so one corner points toward the viewer. Here the ambient light contributed by all three of the lights keeps the colors somewhat muted, but clearly the red light is above, the green light is below and to the right, and the blue light is below and to the left of the viewer's eyepoint. The lights seem to be pastels because each face still gets some of the other two colors from the ambient light; to change this you would need to reduce the ambient light and increase the brightness of the three lights.

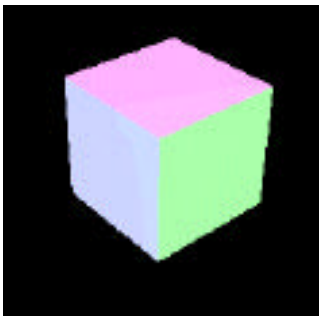


Figure 9.9: the white cube viewed with three colored lights

A word to the wise...

The OpenGL lighting model is essentially the same as the basic lighting model of all standard graphics APIs, but it lacks some very important things that might let you achieve some particular effects you would want if you were to try to get genuine realism in your scenes. One of the most important things lacking in the simple lighting model here is shadows; while OpenGL has techniques that can allow you to create shadows, they are tricky and require some special effort. We will see an example of this when we describe how to create a texture map of shadows by calculating a view from the light point. Another important missing part is the kind of "hot" colors that seem to radiate more of a particular color than they could possibly get in the light they receive, and there is no way to fix this because of the limited gamut of the phosphors in any computer screen, as described in many textbooks. Finally, OpenGL does not allow the kind of directional (anisotropic) reflection that you would need to model materials such as brushed aluminum, which can be created on the computer with special programming. So do not take the OpenGL lighting model as the correct way to do color; take it as a way that works pretty well and that would take much more effort to do better.

Lighting is a seductive effect because it engages our perceptual system to identify shapes of things. This can be very effective, but beware of applying lighting where your shapes or colors are purely arbitrary and represent abstract concepts. It can be dangerous to infer shapes by lighting where

there is no physical reality to the things being displayed. This topic is explored in more detail in the chapter on visual communication.

Shading example

The two issues in using OpenGL shading are the selection of the shading model and the specification of a color at each vertex, either explicitly with the `glColor*(...)` function or by setting a normal per vertex with the `glNormal*(...)` function. The default shading model for OpenGL is smooth, for example, but you will not get the visual effect of smooth shading unless you specify the appropriate normals for your model, as described below. OpenGL allows you to select the shading model with the `glShadeModel` function, and the only values of its single parameter are the symbolic parameters `GL_SMOOTH` and `GL_FLAT`. You may use the `glShadeModel` function to switch back and forth between smooth and flat shading any time you wish.

In the sample code below, we set up smooth shading by the approach of defining a separate normal vector at each vertex. To begin, we will use the following function call in the `init()` fuction to ensure that we automatically normalize all our normals in order to avoid having to do this computation ourselves:

```
glEnable(GL_NORMALIZE);//make unit normals after transforms
```

We use the analytic nature of the surface to generate the normals for each vertex. We compute the partial derivatives $\partial f / \partial x$ and $\partial f / \partial y$ for the function in order to get tangent vectors at each vertex:

```
#define f(x,y)   0.3*cos(x*x+y*y+t)  // original function
#define fx(x,y) -0.6*x*sin(x*x+y*y+t)// partial derivative in x
#define fy(x,y) -0.6*y*sin(x*x+y*y+t)// partial derivative in y
```

In the display function, we first compute the values of `x` and `y` with the functions that compute the grid points in our domain, here called `XX(i)` and `YY(j)`, and then we do the following (fairly long) computation for each triangle in the surface, using an inline cross product operation. We are careful to compute the triangle surface normal as (X–partial cross Y–partial), in that order, so the right-hand rule for cross products gives the correct direction for it.

```
glBegin(GL_POLYGON);
   x = XX(i);
   y = YY(j);
   vec1[0] = 1.0;
   vec1[1] = 0.0;
   vec1[2] = fx(x,y); // partial in X-Z plane
   vec2[0] = 0.0;
   vec2[1] = 1.0;
   vec2[2] = fy(x,y); // partial in Y-Z plane
   Normal[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
   Normal[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
   Normal[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
   glNormal3fv(Normal);
   glVertex3f(XX(i),YY(j),vertices[i][j]);
   ... // do similar code two more times for each vertex of the
   ... // triangle
glEnd();
```

This would probably be handled more efficiently by setting up the `vec1` and `vec2` vectors as above and then calling a utility function to calculate the cross product.

Questions

This set of questions covers your recognition of issues in lighting and shading as you see them in your environment. These will help you see the different kinds of light used in the OpenGL simple local lighting model, and will also help you understand some of the limitations of this model.

1. In your environment, identify examples of objects that show only ambient light, that show diffuse light, and that show specular light. Note the relationship of these objects to direct light sources, and draw conclusions about the relationships that give only ambient light, that give both ambient and diffuse light, and that give specular light. Observe the specular light and see whether it has the color of the object or the color of the light source.

2. In your environment, identify objects that show high, moderate, and low specularity. What seems to be the property of the materials of these objects that makes them show specular light?

3. In your environment, find examples of positional lights and directional lights, and discuss how that would affect your choice of these two kinds of lights in a scene.

4. In your environment, select some objects that seem to be made of different materials and identify the ambient, diffuse, and specular properties of each. Try to define each of these materials in terms of the OpenGL material functions.

5. In your environment, find examples where the ambient lighting seems to be different in different places. What are the contributing factors that make this so? What does this say about the accuracy of local lighting models as compared with global lighting models?

Exercises

This set of exercises asks you to calculate some of things that are important in modeling or lighting, and then often to draw conclusions from your calculations. The calculations should be straightforward based on information from the chapter.

6. If you have a triangle whose vertices are V0 = (, ,), V1 = (, ,), and V2 = (, ,), in that sequence, calculate the appropriate edge vectors and then calculate the unit normal to this triangle based on a cross product of the edge vectors.

7. If you have a surface given by a function of two variables, <fill in function here>, calculate the partial derivatives in *u* and *v*, and for the point P = (, ,), calculate the two directional tangents and, using the cross product, calculate the unit normal to the surface at P.

8. For some particular surfaces, the surface normal is particularly easy to compute. Show how you can easily calculate the normals for a plane, a sphere, or a cylinder.

9. Going back to question 1 above, switch the values of vertices V0 and V2 and do the calculations again. What do you observe about relation between the new unit normal you have just computed and the unit normal you computed originally?

10. We saw in questions 1 and 4 that the orientation of a polygon defines the direction of a surface normal, and the direction of the surface normal affects the sign of the dot product in the diffuse and specular equations. For the normals of questions 1 and 4, assume that you have a light at point L = (, ,), and calculate the diffuse and specular light components for each of the two orientations. Which orientation makes sense, and what does that tell you about the way you define the sequences of vertices for your geometry?

11. In the equation for diffuse light, assume a light has unit energy and calculate the energy reflected at different angles from the normal using the diffuse lighting equation. Similarly, calculate the area of a unit square of surface when it is projected onto a plane at different angles to the surface. Then calculate the ratio of the energy to the projected surface area; this should be a constant. Based on your calculations, why is this so?

12. Based on the specular light equation, write the equation that would let you find the angle Θ at which the energy of the specular light would be 50% of the original light energy. How does this equation vary with the specularity coefficient (the shininess value) N?

13. In exploring the effect of dropoff for spotlights, define spotlight with a cutoff of 45° and constant attenuation, and calculate the light energy over 5° increments from 0° to 45° with dropoff exponents 1, 2, and 4. What does this tell you about the central and edge energies of spotlights?

14. In exploring the effect of attenuation for lights, consider the OpenGL equation for attenuation: $A = 1/(A_C + A_L \quad D + A_Q \quad D^2)$. Define a set of points at distances 1, 2, 3, 4, and 5 units from a light, and calculate the light energy at each of these points using unit coefficients, based on the equation in three cases: (a) only constant attenuation, (b) only linear attenuation, and (c) only quadratic attenuation. What conclusions do you draw about the effect of linear and quadratic attenuation as you observe positional lights in the world around you?

15. For the merry-go-round scene graph defined in the questions in Chapter 2 on modeling, show how you would place a light in the scene graph if the light were (a) at the center of the merry-go-round, (b) on a post at the outside of the base of the merry-go-round, or (c) on top of the head of a horse on the merry-go-round.

16. True or false: for a model with lighting, flat shading is exactly the same as smooth shading if the smooth-shaded model uses the face normal at each vertex. Why is your answer correct?

17. If a triangle is very small, that is, if the number of pixels in the rendered triangle is small, does it really matter whether you use flat or smooth shading for the triangle? Why or why not?

Experiments

Define a hemisphere from scratch using spherical coordinates or any other modeling techniques from Chapter 2 or Chapter 3. Design the hemisphere so you can easily control the resolution of the figure in all dimensions, just as the GLU and GLUT models let you define slices and stacks. This hemisphere will be the basis of the questions below.

18. From the question above in which you were asked to try to define material properties that matched some real-world materials, use these material definitions in the image of the hemisphere and see how closely the image approximates the actual appearance of the object. If the approximation is not very good, see if you can identify why this is so and modify the material definitions to make it better.

19. Because we are working with a hemisphere, it is simple to get analytic normals for each vertex. However, a normal for a face is not the same as the normal for a vertex; show how you can get a normal for each face of the hemisphere.

20. Using routine code for displaying polygonal models and using a relatively coarse definition of the hemisphere, compare the effect of flat shading (using the face normal) and the effect of smooth shading (using vertex normals). Compare the effect of using coarsely modeled smooth shading with using flat shading with much higher resolution.

21. For the display of flat shading, systematically choose a vertex normal for each face instead of the face normal and compare the view you get each way. Why is the vertex normal view less accurate than the face normal view? Is the difference important?

22. Because the display code allows us to rotate the hemisphere and see the inside as well as the outside, we can consider different light models:
    GL_LIGHT_MODEL_TWO_SIDE
    GL_LIGHT_MODEL_LOCAL_VIEWER
    and different face parameters for materials:
    GL_FRONT
    GL_BACK
    GL_FRONT_AND_BACK
    Try out all or most of these options, and for each, note the effect of the option on the view.

Chapter 10: Events and Event Handling for Computer Graphics

Introduction

Graphics programming can focus entirely on creating one single image based on a set of data, but more and more we are seeing the value of writing programs that allow the user to interact with the world through graphical presentations, or that allow the user to control the way an image is created. These are called interactive computer graphics programs, and the ability to interact with information through an image is critically important to the success of this field.

Our emphasis in this chapter is on graphical interaction, not on user interfaces. Certainly many user interfaces use graphical presentations that give information to the user, take graphical actions, and interpret the results for program control, but we simply view these as applications of our graphics. Late in the chapter we introduce the MUI (Micro User Interface) system that allows you to add a primitive interface to your OpenGL programs, and we believe that you should try to understand the nature of the communication about images that can be supported by an external user interface, but a genuine discussion of user interfaces is much too deep for us to undertake here. In general, we subscribe to the view that computer interaction should be designed by persons who are specially trained in human factors, interface design, and evaluation, and not by computer scientists, but computer scientists will implement the design. This chapter and the chapter on selection describe how such implementation can be done.

Interactive programming in computer graphics generally takes advantage of the event-handling capabilities of modern systems, so we must understand something of what events are and how to use them in order to write interactive graphics programs. Events are fairly abstract and come in several varieties, so we will need to go into some details as we develop this idea below. But modern graphics APIs handle events pretty cleanly, and you will find that once you are used to the idea, it is not particularly difficult to write event-driven programs. You should realize that some basic APIs do not include event handling on their own, so you may need to use an extension to the API for this.

Definitions

An *event* is a transition in the control state of a computer system. Events can come from many sources and can cause any of a number of actions to take place as the system responds to the transition. In general, we will treat an event as an abstraction, a concept that we use to design interactive applications, that provides a concrete piece of data to the computer system. An *event record* is a formal record of some system activity, often an activity from a device such as a keyboard or mouse. An event record is a data structure that contains information that identifies the event and any data corresponding to the event. This is not a user-accessible data structure, but its values are returned to the system and application by appropriate system functions. A keyboard event record contains the identity of the key that was pressed and the location of the cursor when it was pressed, for example; a mouse event record contains the mouse key that was pressed, if any, and the cursor's location on the screen when the event took place. Event records are stored in the *event queue*, which is managed by the operating system; this keeps track of the sequence in which events happen and serves as a resource to processes that deal with events. When an event occurs, its event record is inserted in the event queue and we say that the event is *posted* to the queue. The operating system manages the event queue and as each event gets to the front of the queue and a process requests an event record, the operating system passes the record to the process that should handle it. In general, events that involve a screen location get passed to whatever program owns that location, so if the event happens outside a program's window, that program will not get the event.

Let's consider a straightforward example of a user action that causes an event in an application, and think about what is actually done to get and manage that event. This will vary from system to system, so we will consider two alternatives. In the first, the application includes a system utility that polls for events and, when one occurs in the system, identifies the event and calls an appropriate event handler. In the second, the application initializes an event loop and provides event handlers that act like callbacks to respond to events as the system gets them. A graphics API might use either model, but we will focus on the event loop and callback model.

Programs that use events for control—and most interactive programs do this—manage that control through functions that are called *event handlers*. While these can gain access to the event queue in a number of ways, most APIs use functions called *callbacks* to handle events. Associating a callback function with an event is called *registering* the callback for the event. When the system passes an event record to the program, the program determines what kind of event it is and if any callback function has been registered for the event, passes control to that function. In fact, most interactive programs contain initialization and action functions, callback functions, and a *main event loop*. The main event loop invokes an event handler whose function is to get an event, determine the callback needed to handle the event, and pass control to that function. When that function finishes its operation, control is returned to the event handler.

What happens in the main event loop is straightforward—the program gives up direct control of the flow of execution and places it in the hands of the user. From this point throughout the remainder of the execution of the program, the user will cause events to which the program will respond through the callbacks that you have created. We will see many examples of this approach in this, and later, chapters.

A callback is a function that is executed when a particular event is recognized by the program. This recognition happens when the event handler takes an event off the event queue and the program has *expressed an interest* in the event. The key to being able to use a certain event in a program, then, is to express an interest in the event and to indicate what function is to be executed when the event happens—the function that has been registered for the event.

Some examples of events

Events are often categorized in a set of classes, depending on the kind of action that causes the event. Below we describe one possible way of classifying events that gives you the flavor of this concept.

keypress events, such as keyDown, keyUp, keyStillDown, ... Note that there may be two different kinds of keypress events: those that use the regular keyboard and those that use the so-called "special keys" such as the function keys or the cursor control keys. There may be different event handlers for these different kinds of keypresses. You should be careful when you use special keys, because different computers may have different special keys, and those that are the same may be laid out in different ways.

mouse events, such as leftButtonDown, leftButtonUp, leftButtonStillDown, … Note that different "species" of mice have different numbers of buttons, so for some kinds of mice some of these events are collapsed.

menu events, such as selection of an item from a pop-up or pull-down menu or submenu, that are based on menu choices.

window events, such as moving or resizing a window, that are based on standard window manipulations.

system events, such as idle and timer, that are generated by the system based on the state of the event queue or the system clock, respectively.

software events, which are posted by programs themselves in order to get a specific kind of processing to occur next.

These events are very detailed, and many of them are not used in the APIs or API extensions commonly found with graphics. However, all could be used by going deeply enough into the system on which programs are being developed.

Note that event-driven actions are fundamentally different from actions that are driven by polling—that is, by querying a device or some other part of the system on some schedule and basing system activity on the results. There are certainly systems that operate by polling various kinds of input and interaction devices, but these are outside our current approach.

The vocabulary of interaction

When users are working with your application, they are focusing on the content of their work, not on how you designed the application. They want to be able to communicate with the program and their data in ways that feel natural to them, and it is the task of the interface designer to create an interface that feels very natural and that doesn't interfere with their work. Interface design is the subject of a different course from computer graphics, but it is useful to have a little understanding of the vocabulary of interaction.

We have been focusing on how to program interaction with the kind of devices that are commonly found in current computers: keyboards or mice. These devices have distinctly different kinds of behaviors in users' minds. When you get away from text operations, keyboards give discrete input that can be interpreted in different ways depending on the keys that are pressed. They are basically devices that make abstract selections, with the ability select actions as well as objects. The keyboard input that navigates through simple text games is an example of action selection. The mouse buttons are also selection devices, although they are primarily used to select graphical objects on the screen, including control buttons as well as displayed objects. The keyboard and mouse buttons both are discrete devices, providing only a finite number of well-defined actions.

The mouse itself has a different kind of meaning. It provides a more continuous input, and can be used to control continuous motion on the screen. This can be the motion of a selected object as it is moved into a desired position, or it can be an input that will cause motion in an object. The motion that the mouse controls can be of various kinds as well—it can be a linear motion, such as moving the eye point across a scene, or it can be a rotational motion, such as moving an object by changing the angles defining the object in spherical coordinates.

When you plan the interaction for your application, then, you should decide whether a user will see the interaction as a discrete selection or as a continuous control, and then you should implement the interaction with the keyboard or mouse, as determined by the user's expected vocabulary.

A word to the wise...

This section discusses the mechanics of interaction through event handling, but it does not cover the critical questions of how a user would naturally control an interactive application. There are many deep and subtle issues involved in designing the user interface for such an application, and this module does not begin to cover them. The extensive literature in user interfaces will help you get a start in this area, but a professional application needs a professional interface, one designed, tested, and evolved by persons who focus in this area. When thinking of a real application, heed the old cliché: Kids, don't try this at home!

The examples below do their best to present user controls that are not impossibly clumsy, but they are designed much more to focus on the event and callback than on a clever or smooth way for a user to work. When you write your own interactive projects, think carefully about how a user might perceive the task, not just about an approach that might be easiest for you to program.

Events in OpenGL

The OpenGL API generally uses the Graphics Library Utility Toolkit GLUT (or a similar extension) for event and window handling. GLUT defines a number of kinds of events and gives the programmer a means of associating a callback function with each event that the program will use. In OpenGL with the GLUT extension, this main event loop is quite explicit as a call to the function `glutMainLoop()` as the last action in the main program.

Callback registering

Below we will list some kinds of events and will then indicate the function that is used to register the callback for each event. Following that, we will give some code examples that register and use these events for some programming effects. This now includes only examples from OpenGL, but it should be extensible to other APIs fairly easily.

Event	Callback Registration Function

idle	`glutIdleFunc(functionname)`
	requires a callback function with template
	`void functionname(void)`
	as a parameter. This function is the event handler that determines what is to be done at each idle cycle. Often this function will end with a call to `glutPostRedisplay()` as described below. This function is used to define what action the program is to take when there has been no other event to be handled, and is often the function that drives real-time animations.

display	`glutDisplayFunc(functionname)`
	requires a callback function with template
	`void functionname(void)`
	as a parameter. This function is the event handler that generates a new display whenever the display event is received. Note that the display function is invoked by the event handler whenever a display event is reached; this event is posted by the `glutPostRedisplay()` function and whenever a window is moved or reshaped.

reshape	`glutReshapeFunc(functionname)`
	requires a callback function with template
	`void functionname(int, int)`
	as a parameter. This function manages any changes needed in the view setup to accomodate the reshaped window, which may include a fresh definition of the projection. The parameters of the `reshape` function are the width and height of the window after it has been changed.

keyboard	`glutKeyboardFunc(keybd)`
	requires a callback function with template
	`void functionname(unsigned char, int, int)`
	as a parameter. This parameter function is the event handler that receives the character and the location of the cursor `(int x, int y)` when a key

is pressed. As is the case for all callbacks that involve a screen location, the location on the screen will be converted to coordinates relative to the window. Again, this function will often end with a call to `glutPostRedisplay()` to re-display the scene with the changes caused by the particular keyboard event.

special

`glutSpecialFunc(special)`
requires a callback function with template

```
void functionname(int key, int x, int y)
```

as a parameter. This event is generated when one of the "special keys" is pressed; these keys are the function keys, directional keys, and a few others. The first parameter is the key that was pressed; the second and third are the integer window coordinates of the cursor when the keypress occurred as described above. The usual approach is to use a special symbolic name for the key, and these are described in the discussion below. The only difference between the special and keyboard callbacks is that the events come from different kinds of keys.

menu

`int glutCreateMenu(functionname)`
requires a callback function with template

```
void functionname(int)
```

as a parameter. The integer value passed to the function is the integer assigned to the selected menu choice when the menu is opened and a choice is made; below we describe how menu entries are associated with these values.

The `glutCreateMenu()` function returns a value that identifies the menu for later operations that can change the menu choices. These operations are discussed later in this chapter when we describe how menus can be manipulated. The `glutCreateMenu()` function creates a menu that is brought up by a mouse button down event, specified by

```
glutAttachMenu(event),
```

which attaches the current menu to an identified event, and the function

```
glutAddMenuEntry(string, int)
```

identifies each of the choices in the menu and defines the value to be returned by each one. That is, when the user selects the menu item labeled with the string, the value is passed as the parameter to the menu callback function. The menu choices are identified before the menu itself is attached, as illustrated in the lines below:

```
glutAddMenuEntry("text", VALUE);
...
glutAttachMenu(GLUT_RIGHT_BUTTON)
```

The `glutAttachMenu()` function signifies the end of creating the menu.

Note that the Macintosh uses a slightly different menu attachment with the same parameters,

```
glutAttachMenuName(event, string),
```

that attaches the menu to a name on the system menu bar. The Macintosh menu is activated by selecting the menu name from the menu bar, while the windows for Unix and Windows are popup windows that appear where the mouse is clicked and that do not have names attached.

Along with menus one can have sub-menus—items in a menu that cause a cascaded sub-menu to be displayed when it is selected. Sub-menus are

created two ways; here we describe adding a sub-menu by using the function

```
glutAddSubMenu(string, int)
```

where the string is the text displayed in the original menu and the integer is the identifier of the menu to cascade from that menu item. When the string item is chosen in the original menu, the submenu will be displayed. With this GLUT function, you can only add a sub-menu as the last item in a menu, so adding a sub-menu closes the creation of the main menu. However, later in this chapter we describe how you can add more submenus within a menu.

mouse

```
glutMouseFunc(functionname)
```

requires a callback function with a template such as

```
void functionname(int button, int state,
int mouseX, int mouseY)
```

as a parameter, where `button` indicates which button was pressed (an integer typically made up of one bit per button, so that a three-button mouse can indicate any value from one to seven), the `state` of the mouse (symbolic values such as `GLUT_DOWN` to indicate what is happening with the mouse) — and both raising and releasing buttons causes events — and integer values `xPos` and `yPos` for the window-relative location of the cursor in the window when the event occurred.

The mouse event does not use this function if it includes a key that has been defined to trigger a menu.

mouse active motion

```
glutMotionFunc(functionname)
```

requires a callback function with template

```
void functionname(int, int)
```

as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with one or more buttons pressed.

mouse passive motion

```
glutPassiveMotionFunc(functionname)
```

requires a callback function with template

```
void functionname(int, int)
```

as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse if moved with no buttons pressed.

timer

```
glutTimerFunc(msec, timer, value)
```

requires an integer parameter, here called `msec`, that is to be the number of milliseconds that pass before the callback is triggered; a callback function, here called `timer`, with a template such as

```
void timer(int)
```

that takes an integer parameter; and an integer parameter, here called `value`, that is to be passed to the `timer` function when it is called.

Note that in any of these cases, the function NULL is an acceptable option. Thus you can create a template for your code that includes registrations for all the events your system can support, and simply register the NULL function for any event that you want to ignore.

Besides the kind of device events we generally think of, there are also software events such as the display event, created by a call to `glutPostRedisplay()`. There are also device events for devices that are probably not found around most undergraduate laboratories: the spaceball, a six-degree-of-freedom deviceused in high-end applications, and the graphics tablet, a device familiar to the computer-aided design world and still valuable in many applications. If you want to know more about handling these devices, you should check the GLUT manual.

Some details

For most of these callbacks, the meaning of the parameters of the event callback is pretty clear. Most are either standard characters or integers such as window dimensions or cursor locations. However, for the special event, the callback must handle the special characters by symbolic names. Many of the names are straightforward, but some are not; the full table is:

Function keys F1 through F12:	GLUT_KEY_F1 through GLUT_KEY_F12
Directional keys:	GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN
Other special keys:	GLUT_KEY_PAGE_UP (Page up)
	GLUT_KEY_PAGE_DOWN (Page down)
	GLUT_KEY_HOME (Home)
	GLUT_KEY_END (End)
	GLUT_KEY_INSERT (Insert)

So to use the special keys, use these symbolic names to process the keypress that was returned to the callback function.

More on menus

Earlier in the chapter we saw how we could create menus, add menu items, and specify the menu callback function. But menus are complex resources that can be managed with much more detail than this. Menus can be activated and deactivated, can be created and destroyed, and menu items can be added, deleted, or modified. The basic tools to do this are included in the GLUT toolkit and are described in this section.

You will have noticed that when you define a menu, the `glutCreateMenu()` function returns an integer value. This value is the menu number. While the menu you are creating is the active menu while you are creating it, if you have more than one menu you will have to refer to a particular menu by its number when you operate on it. In order to see what the active menu number is at any point, you can use the function

```
int glutGetMenu(void)
```

that simply returns the menu number. If you need to change the active menu at any point, you can do so by using its number as the argument to the function

```
void glutSetMenu(int menu)
```

This will make the menu whose number you choose into the active menu so the operations we describe below can be done to it. Note that both main menus and sub-menus have menu numbers, so it is important to keep track of them.

It is also possible to detach a menu from a button, in effect deactivating the menu. This is done by the function

```
void glutDetachMenu(event)
```

which detaches the event from the current menu.

Menus can be dynamic. You can change the string and the returned value of any menu entry with the function

```
void glutChangeToMenuEntry(int entry, char *name, int value)
```
where the name is the new string to be displayed and the new value is the value that the event handler is to return to the system when this item is chosen. The menu that will be changed is the active window, which can be set as described above.

While you may only create one sub-menu to a main menu with the `glutAddSubMenu()` function we described above, you may add sub-menus later by using the
```
void glutChangeToSubMenu(int entry, char *name, int menu)
```
function. Here the entry is the number in the current menu (the first item is numbered 1) that is to be changed into a submenu trigger, the name is the string that is to be displayed at that location, and menu is the number to be given to the new sub-menu. This will allow you to add sub-menus to any menu you like at any point you like.

Menus can also be destroyed as well as attached and detached. The function
```
void glutDestroyMenu(int menu)
```
destroys the menu whose identifier is passed as the parameter to the function.

These details can seem overwhelming until you have a reason to want to change menus as your program runs. When you have a specific need to make changes in your menus, you will probably find that the GLUT toolkit has enough tools to let you do the job.

Code examples

This section presents four examples. This first is a simple animation that uses an idle event callback and moves a cube around a circle, in and out of the circle's radius, and up and down. The user has no control over this motion. When you compile and run this piece of code, see if you can imagine the volume in 3-space inside which the cube moves.

The second example uses keyboard callbacks to move a cube up/down, left/right, and front/back by using a simple keypad on the keyboard. This uses keys within the standard keyboard instead of using special keys such as a numeric keypad or the cursor control keys. A numeric keypad is not used because some keyboards do not have them; the cursor control keys are not used because we need six directions, not just four.

The third example uses a mouse callback to pop up a menu and make a menu selection, in order to set the color of a cube. This is a somewhat trivial action, but it introduces the use of pop-up menus, which are a very standard and useful tool.

Finally, the fourth example uses a mouse callback with object selection to identify one of two cubes that are being displayed and to change the color of that cube. Again, this is not a difficult action, but it calls upon the entire selection buffer process that is the subject of another later module in this set. For now, we suggest that you focus on the event and callback concepts and postpone a full understanding of this example until you have read the material on selection.

Idle event callback

In this example, we assume we have a function named `cube()` that will draw a simple cube at the origin `(0,0,0)`. We want to move the cube around by changing its position with time, so we will let the idle event handler set the position of the cube and the display function draw the cube using the positions determined by the idle event handler. Much of the code for a complete program has been left out, but this illustrates the relation between the display function, the event handler, and the callback registration.

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void )
{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void animate(void)
{
    #define deltaTime 0.05

// Position for the cube is set by modeling time-based behavior.
// Try multiplying the time by different constants to see how that
// behavior changes.

    time += deltaTime; if (time > 2.0*M_PI) time -= 2*0*M_PI;
    cubex = sin(time);
    cubey = cos(time);
    cubez = cos(time);
    glutPostRedisplay();
}

void main(int argc, char** argv)
{

/* Standard GLUT initialization precedes the functions below*/
    ...
    glutDisplayFunc(display);
    glutIdleFunc(animate);

    myinit();
    glutMainLoop();
}
```

Keyboard callback

Again we start with the familiar `cube()` funcntion. This time we want to let the user move the cube up/down, left/right, or backward/forward by means of simple keypresses. We will use two virtual keypads:

```
Q  W            I  O
 A  S          J  K
  Z  X        N  M
```

with the top row controlling up/down, the middle row controlling left/right, and the bottom row controlling backward/forward. So, for example, if the user presses either `Q` or `I`, the cube will move up; pressing `W` or `O` will move it down. The other rows will work similarly.

Again, much of the code has been omitted, but the display function works just as it did in the example above: the event handler sets global positioning variables and the display function

performs a translation as chosen by the user. Note that in this example, these translations operate in the direction of faces of the cube, not in the directions relative to the window.

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void )
{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void keyboard(unsigned char key, int x, int y)
{
    ch = ' ';
    switch (key)
    {
        case 'q' : case 'Q' :
        case 'i' : case 'I' :
           ch = key; cubey -= 0.1; break;
        case 'w' : case 'W' :
        case 'o' : case 'O' :
           ch = key; cubey += 0.1; break;
        case 'a' : case 'A' :
        case 'j' : case 'J' :
           ch = key; cubex -= 0.1; break;
        case 's' : case 'S' :
        case 'k' : case 'K' :
           ch = key; cubex += 0.1; break;
        case 'z' : case 'Z' :
        case 'n' : case 'N' :
           ch = key; cubez -= 0.1; break;
        case 'x' : case 'X' :
        case 'm' : case 'M' :
           ch = key; cubez += 0.1; break;
    }
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
/* Standard GLUT initialization */
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    myinit();
    glutMainLoop();
}
```

The similar function, `glutSpecialFunc(...)`, can be used in a very similar way to read input from the special keys (function keys, cursor control keys, ...) on the keyboard.

Menu callback

Again we start with the familiar `cube()` function, but this time we have no motion of the cube. Instead we define a menu that allows us to choose the color of the cube, and after we make our choice the new color is applied.

```
#define RED     1
#define GREEN   2
#define BLUE    3
#define WHITE   4
#define YELLOW  5

void cube(void)
{
    ...

    GLfloat color[4];

//  set the color based on the menu choice

    switch (colorName) {
        case RED:
            color[0] = 1.0; color[1] = 0.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case GREEN:
            color[0] = 0.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case BLUE:
            color[0] = 0.0; color[1] = 0.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case WHITE:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case YELLOW:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
    }

//  draw the cube

    ...
 }

void display( void )
{
    cube();
}

void options_menu(int input)
{
    colorName = input;
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
    ...

        glutCreateMenu(options_menu);        // create options menu
```

```
        glutAddMenuEntry("Red", RED);        // 1 add menu entries
        glutAddMenuEntry("Green", GREEN);    // 2
        glutAddMenuEntry("Blue", BLUE);      // 3
        glutAddMenuEntry("White", WHITE);    // 4
        glutAddMenuEntry("Yellow", YELLOW);  // 5
        glutAttachMenu(GLUT_RIGHT_BUTTON, "Colors");

        myinit();
        glutMainLoop();
}
```

Mouse callback for object selection

This example is more complex because it illustrates the use of a mouse event in object selection. This subject is covered in more detail in the later chapter on object selection, and the full code example for this example will also be included there. We will create two cubes with the familiar `cube()` function, and we will select one with the mouse. When we select one of the cubes, the cubes will exchange colors.

In this example, we start with a full `Mouse(...)` callback function, the `render(...)` function that registers the two cubes in the object name list, and the `DoSelect(...)` function that manages drawing the scene in `GL_SELECT` mode and identifying the object(s) selected by the position of the mouse when the event happened. Finally, we include the statement in the `main()` function that registers the mouse callback function.

```
glutMouseFunc(Mouse);

...

void Mouse(int button, int state, int mouseX, int mouseY)
{
   if (state == GLUT_DOWN) { /* find which object was selected */
         hit = DoSelect((GLint) mouseX, (GLint) mouseY);
   }
   glutPostRedisplay();
}

...

void render( GLenum mode )
{
// Always draw the two cubes, even if we are in GL_SELECT mode,
// because an object is selectable iff it is identified in the name
// list and is drawn in GL_SELECT mode
   if (mode == GL_SELECT)
         glLoadName(0);
   glPushMatrix();
   glTranslatef(  1.0,  1.0, -2.0 );
   cube(cubeColor2);
   glPopMatrix();
   if (mode == GL_SELECT)
         glLoadName(1);
   glPushMatrix();
    glTranslatef( -1.0, -2.0,  1.0 );
   cube(cubeColor1);
   glPopMatrix();
   glFlush();
```

```
   glutSwapBuffers();
}

...

GLint DoSelect(GLint x, GLint y)
{
   GLint hits, temp;

   glSelectBuffer(MAXHITS, selectBuf);
   glRenderMode(GL_SELECT);
   glInitNames();
   glPushName(0);

// set up the viewing model
   glPushMatrix();
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
// set up the matrix that identifies the picked object(s), based on
// the  x  and  y  values of the selection and the information on the
// viewport
   gluPickMatrix(x, windH - y, 4, 4, vp);
   glClearColor(0.0, 0.0, 1.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   gluPerspective(60.0,1.0,1.0,30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
//               eye point     center of view      up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   render(GL_SELECT);  // draw the scene for selection

   glPopMatrix();
// find the number of hits recorded and reset mode of render
   hits = glRenderMode(GL_RENDER);
// reset viewing model into GL_MODELVIEW mode
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,1.0,1.0,30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
//               eye point     center of view      up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// return the label of the object selected, if any
   if (hits <= 0) {
         return -1;
   }
// carry out the color changes that will be the effect of a selection
   temp = cubeColor1; cubeColor1 = cubeColor2; cubeColor2 = temp;
   return selectBuf[3];
}

void main(int argc, char** argv)
{
   ...
   glutMouseFunc(Mouse);
   myinit();
   glutMainLoop();
}
```

Mouse callback for mouse motion

This example shows the callback for the motion event. This event can be used for anything that uses the position of a moving mouse with button pressed as control. It is fairly common to see a graphics program that lets the user hold down the mouse and drag the cursor around in the window, and the program responds by moving or rotating the scene around the window. The program this code fragment is from uses the integer coordinates to control spin, but they could be used for many purposes and the application code itself is omitted.

```
void motion(int xPos, int yPos)
{
   spinX = (GLfloat)xPos;
   spinY = (GLfloat)yPos;
}

int main(int argc, char** argv)
{
   ...
   glutMotionFunc(motion);

   myinit();
   glutMainLoop();
}
```

The MUI (Micro User Interface) facility

Prerequisites

An understanding of event-driven programming and some experience using the simple events and callbacks from the GLUT toolkit in OpenGL, and some review of interface capabilities in standard applications.

Introduction

There are many kinds of interface tools that we are used to seeing in applications but that we cannot readily code in OpenGL, even with the GLUT toolkit. Some of these are provided by the MUI facility that is a universal extension of GLUT for OpenGL. With MUI you can use sliders, buttons, text boxes, and other tools that may be more natural for many applications than the standard GLUT capabilities. Of course, you may choose to write your own tools as well, but you may choose to use your time on the problem at hand instead of writing an interface, so the MUI tools may be just what you want.

MUI has a good bit of the look and feel of the X-Motif interface, so do not expect applications you write with this to look like they are from either the Windows or Macintosh world. Instead, focus on the functionality you need your application to have, and find a way to get this functionality from the MUI tools. The visible representation of these tools are called *widgets*, just as they are in the X Window System, so you will see this term throughout thest notes.

This chapter is built on Steve Baker's "A Brief MUI User Guide," and it shares similar properties: it is based on a small number of examples and some modest experimental work. It is intended as a guide, not as a manual, though it is hoped that it will contribute to the literature on this useful tool.

Definitions

The capabilities of MUI include pulldown menus, buttons, radio buttons, text labels, text boxes, and vertical and horizontal sliders. We will outline how each of these work below and will include some general code to show how each is invoked.

The main thing you must realize in working with MUI is that MUI takes over the event handling from GLUT, so you cannot mix MUI and GLUT event-handling capabilities in the same window. This means that you will have to create separate windows for your MUI controls and for your display, which can feel somewhat clumsy. This is a tradeoff you must make when you design your application — are you willing to create a different kind of interface than you might expect in a traditional application in order to use the extra MUI functionality? Only you can say. But before you can make that choice, you need to know what each of the MUI facilities can do.

Menu bars: A MUI menu bar is essentially a GLUT menu that is bound to a MUI object and then that object is added to a UIlist. Assuming you have defined an array of GLUT menus named `myMenus[...]`, you can use the function to create a new pulldown menu and then use the function to add new menus to the pulldown menu list:

```
muiObject *muiNewPulldown();
muiAddPulldownEntry(muiObject *obj,char *title,int glut_menu,
                    int is_help);
```

An example of the latter function would be

```
myMenubar = muiNewPulldown();
muiAddPulldownEntry(myMenubar, "File", myMenu, 0);
```

where the is_help value would be 1 for the last menu in the menu bar, because traditionally the help menu is the rightmost menu in a menu bar.

According to Baker [Bak], there is apparently a problem with the pulldown menus when the GLUT window is moved or resized. The reader is cautioned to be careful in handling windows when the MUI facility is being used.

Buttons: a button is presented as a rectangular region which, when pressed, sets a value or carries out a particular operation. Whenever the cursor is in the region, the button is highlighted to show that it is then selectable. A button is created by the function

```
muiNewButton(int xmin, int xmax, int ymin, int ymax)
```

that has a `muiObject *` return value. The parameters define the rectangle for the button and are defined in window (pixel) coordinates, with `(0,0)` at the lower left corner of the window. In general, any layout in the MUI window will be based on such coordinates.

Radio buttons: radio buttons are similar to standard buttons, but they come in only two fixed sizes (either a standard size or a mini size). The buttons can be designed so that more than one can be pressed (to allow a user to select any subset of a set of options) or they can be linked so that when one is pressed, all the others are un-pressed (to allow a user to select only one of a set of options). Like regular buttons, they are highlighted when the cursor is scrolled over them.

You create radio buttons with the functions

```
muiObject *muiNewRadioButton(int xmin, int ymin)
muiObject *muiNewTinyRadioButton(int xmin, int ymin)
```

where the `xmin` and `ymin` are the window coordinates of the lower left corner of the button. The buttons are linked with the function

```
void muiLinkButtons(button1, button2)
```

where `button1` and `button2` are the names of the button objects; to link more buttons, call the function with overlapping pairs of button names as shown in the example below. In order to clear all the buttons in a group, call the function below with any of the buttons as a parameter:

```
void muiClearRadio(muiObject *button)
```

Text boxes: a text box is a facility to allow a user to enter text to the program. The text can then be used in any way the application wishes. The text box has some limitations; for example, you cannot enter a string longer than the text box's length. However, it also gives your user the ability to enter text and use backspace or delete to correct errors. A text box is created with the function

```
muiObject *muiNewTextbox(xmin, xmax, ymin)
```

whose parameters are window coordinates, and there are functions to set the string:

```
muiSetTBString(obj, string)
```

to clear the string:

```
muiClearTBString(obj)
```

and to get the value of the string:

```
char *muiGetTBString (muiObject *obj).
```

Horizontal sliders: in general, sliders are widgets that return a single value when they are used. The value is between zero and one, and you must manipulate that value into whatever range your application needs. A slider is created by the function

```
muiNewHSlider(int xmin,int ymin,int xmax,int scenter,int shalf)
```

where `xmin` and `ymin` are the screen coordinates of the lower left corner of the slider, `xmax` is the screen coordinate of the right-hand side of the slider, `scenter` is the screen coordinate of the center of the slider's middle bar, and `shalf` is the half-size of the middle bar itself. In the event callback for the slider, the function `muiGetHSVal(muiObject *obj)` is used to return the value (as a float) from the slider to be used in the application. In order to reverse the process — to make the slider represent a particular value, use the function

```
muiSetHSValue(muiObject *obj, float value)
```

Vertical sliders: vertical sliders have the same functionality as horizontal sliders, but they are aligned vertically in the control window instead of horizontally. They are managed by functions that are almost identical to those of horizontal sliders:

```
muiNewVSlider(int xmin,int ymin,int ymax,int scenter,int shalf)
muiGetVSValue(muiObject *obj, float value)
muiSetVSValue(muiObject *obj, float value)
```

Text labels: a text label is a piece of text on the MUI control window. This allows the program to communicate with the user, and can be either a fixed or variable string. To set a fixed string, use

```
muiNewLabel(int xmin, int ymin, string)
```

with xmin and ymin setting the lower left corner of the space where the string will be displayed. To define a variable string, you give the string a `muiObject` name via the variation

```
muiObject *muiNewLabel(int xmin, int ymin, string)
```

to attach a name to the label, and use the `muiChangeLabel(muiObject *, string)` function to change the value of the string in the label.

Using the MUI functionality

Before you can use any of MUI's capabilities, you must initialize the MUI system with the function `muiInit()`, probably called from the `main()` function as described in the sample code below.

MUI widgets are managed in UI lists. You create a UI list with the `muiNewUIList(int)` function, giving it an integer name with the parameter, and add widgets to it as you wish with the function `muiAddToUIList(listid, object)`. You may create multiple lists and can choose which list will be active, allowing you to make your interface context sensitive. However, UI lists are essentially static, not dynamic, because you cannot remove items from a list or delete a list.

All MUI capabilities can be made visible or invisible, active or inactive, or enabled or disabled. This adds some flexibility to your program by letting you customize the interface based on a particular context in the program. The functions for this are:

```
void muiSetVisible(muiObject *obj, int state);
void muiSetActive(muiObject *obj, int state);
void muiSetEnable(muiObject *obj, int state);
int muiGetVisible(muiObject *obj);
int muiGetActive(muiObject *obj);
int muiGetEnable(muiObject *obj);
```

Figure 11.1 shows most of the MUI capabilities: labels, horizontal and vertical sliders, regular and radio buttons (one radio button is selected and the button is highlighted by the cursor as shown), and a text box. Some text has been written into the text box. This gives you an idea of what the standard MUI widgets look like, but because the MUI source is available, you have the opportunity to customize the widgets if you want, though this is beyond the scope of this discussion. Layout is facilitated by the ability to get the size of a MUI object with the function

```
void muiGetObjectSize(muiObject *obj, int *xmin, int *ymin,
                      int *xmax, int *ymax);
```

Figure 11.1: the set of MUI facilities on a single window

MUI object callbacks are optional (you would probably not want to register a callback for a fixed text string, for example, but you would with an active item such as a button). In order to register a callback, you must name the object when it is created and must link that object to its callback function with

```
void muiSetCallback(muiObject *obj, callbackFn)
```

where a callback function has the structure

```
void callbackFn(muiObject *obj, enum muiReturnValue)
```

Note that this callback function need not be unique to the object; in the example below we define a single callback function that is registered for three different sliders and another to handle three different radio buttons, because the action we need from each is the same; when we need to know which object handled the event, this information is available to us as the first parameter of the callback.

If you want to work with the callback return value, the declaration of the `muiReturnValue` is:

```
enum muiReturnValue {
     MUI_NO_ACTION,
     MUI_SLIDER_MOVE,
     MUI_SLIDER_RETURN,
     MUI_SLIDER_SCROLLDOWN,
     MUI_SLIDER_SCROLLUP,
     MUI_SLIDER_THUMB,
     MUI_BUTTON_PRESS,
     MUI_TEXTBOX_RETURN,
     MUI_TEXTLIST_RETURN,
     MUI_TEXTLIST_RETURN_CONFIRM
};
```

so you can look at these values explicitly. For the example below, the button press is assumed because it is the only return value associated with a button, and the slider is queried for its value instead of handling the actual MUI action.

Some examples

Let's consider a simple application and see how we can create the controls for it using the MUI facility. The application is color choice, commonly handled with three sliders (for R/G/B) or four sliders (for R/G/B/A) depending on the need of the user. This application typically provides a way

to display the color that is chosen in a region large enough to reduce the interference of nearby colors in perceiving the chosen color. The application we have in mind is a variant on this that not only shows the color but also shows the three fixed-component planes in the RGB cube and draws a sphere of the selected color (with lighting) in the cube.

The design of this application is built on an example in the Science Examples chapter that shows three cross-sections of a real function of three variables. In order to determine the position of the cross sections, we use a control built on MUI sliders. We also add radio buttons to allow the user to define the size of the sphere at the intersection of the cross-section slices.

Selected code for this application includes declarations of muiObjects, callback functions for sliders and buttons, and the code in the main program that defines the MUI objects for the program, links them to their callback functions, and adds them to the single MUI list we identify. The main issue is that MUI callbacks, like the GLUT callbacks we met earlier, have few parameters and do most of their work by modifying global variables that are used in the other modeling and rendering operations.

```
// selected declarations of muiObjects and window identifiers
muiObject *Rslider, *Gslider, *Bslider;
muiObject *Rlabel, *Glabel, *Blabel;
muiObject *noSphereB, *smallSphereB, *largeSphereB;
int muiWin, glWin;

// callbacks for buttons and sliders
void readButton(muiObject *obj, enum muiReturnValue rv) {
   if ( obj == noSphereB )
      sphereControl = 0;
   if ( obj == smallSphereB )
      sphereControl = 1;
   if (obj == largeSphereB )
      sphereControl = 2;
  glutSetWindow( glWin );
  glutPostRedisplay();
}

void readSliders(muiObject *obj, enum muiReturnValue rv) {
   char rs[32], gs[32], bs[32];
   glutPostRedisplay();

   rr = muiGetHSVal(Rslider);
   gg = muiGetHSVal(Gslider);
   bb = muiGetHSVal(Bslider);

   sprintf(rs,"%6.2f",rr);
   muiChangeLabel(Rlabel, rs);
   sprintf(gs,"%6.2f",gg);
   muiChangeLabel(Glabel, gs);
   sprintf(bs,"%6.2f",bb);
   muiChangeLabel(Blabel, bs);

   DX = -4.0 + rr*8.0;
   DY = -4.0 + gg*8.0;
   DZ = -4.0 + bb*8.0;

   glutSetWindow(glWin);
   glutPostRedisplay();
}
```

```
void main(int argc, char** argv){
   char rs[32], gs[32], bs[32];
// Create MUI control window and its callbacks
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
   glutInitWindowSize(270,350);
   glutInitWindowPosition(600,70);
   muiWin = glutCreateWindow("Control Panel");
   glutSetWindow(muiWin);
   muiInit();
   muiNewUIList(1);
   muiSetActiveUIList(1);

// Define color control sliders
   muiNewLabel(90, 330, "Color controls");

   muiNewLabel(5, 310, "Red");
   sprintf(rs,"%6.2f",rr);
   Rlabel = muiNewLabel(35, 310, rs);
   Rslider = muiNewHSlider(5, 280, 265, 130, 10);
   muiSetCallback(Rslider, readSliders);

   muiNewLabel(5, 255, "Green");
   sprintf(gs,"%6.2f",gg);
   Glabel = muiNewLabel(35, 255, gs);
   Gslider = muiNewHSlider(5, 225, 265, 130, 10);
   muiSetCallback(Gslider, readSliders);

   muiNewLabel(5, 205, "Blue");
   sprintf(bs,"%6.2f",bb);
   Blabel = muiNewLabel(35, 205, bs);
   Bslider = muiNewHSlider(5, 175, 265, 130, 10);
   muiSetCallback(Bslider, readSliders);

// define radio buttons
   muiNewLabel(100, 150, "Sphere size");
   noSphereB    = muiNewRadioButton(10, 110);
   smallSphereB = muiNewRadioButton(100, 110);
   largeSphereB = muiNewRadioButton(190, 110);
   muiLinkButtons(noSphereB, smallSphereB);
   muiLinkButtons(smallSphereB, largeSphereB);
   muiLoadButton(noSphereB,    "None");
   muiLoadButton(smallSphereB, "Small");
   muiLoadButton(largeSphereB, "Large");
   muiSetCallback(noSphereB,    readButton);
   muiSetCallback(smallSphereB, readButton);
   muiSetCallback(largeSphereB, readButton);
   muiClearRadio(noSphereB);

// add sliders and radio buttons to UI list 1
   muiAddToUIList(1, Rslider);
   muiAddToUIList(1, Gslider);
   muiAddToUIList(1, Bslider);
   muiAddToUIList(1, noSphereB);
   muiAddToUIList(1, smallSphereB);
   muiAddToUIList(1, largeSphereB);

// Create display window and its callbacks
   ...
}
```

The presentation and communication for this application are shown in Figure 11.2 below. As the sliders set the R, G, and B values for the color, the numerical values are shown above the sliders and the three planes of constant R, G, and B are shown in the RGB cube. At the intersection of the three planes is drawn a sphere of the selected color in the size indicated by the radio buttons. The RGB cube itself can be rotated by the usual keyboard controls so the user can compare the selected color with nearby colors in those planes, but you have the usual issues of active windows: you must make the display window active to rotate the cube, but you must make the control window active to use the controls.

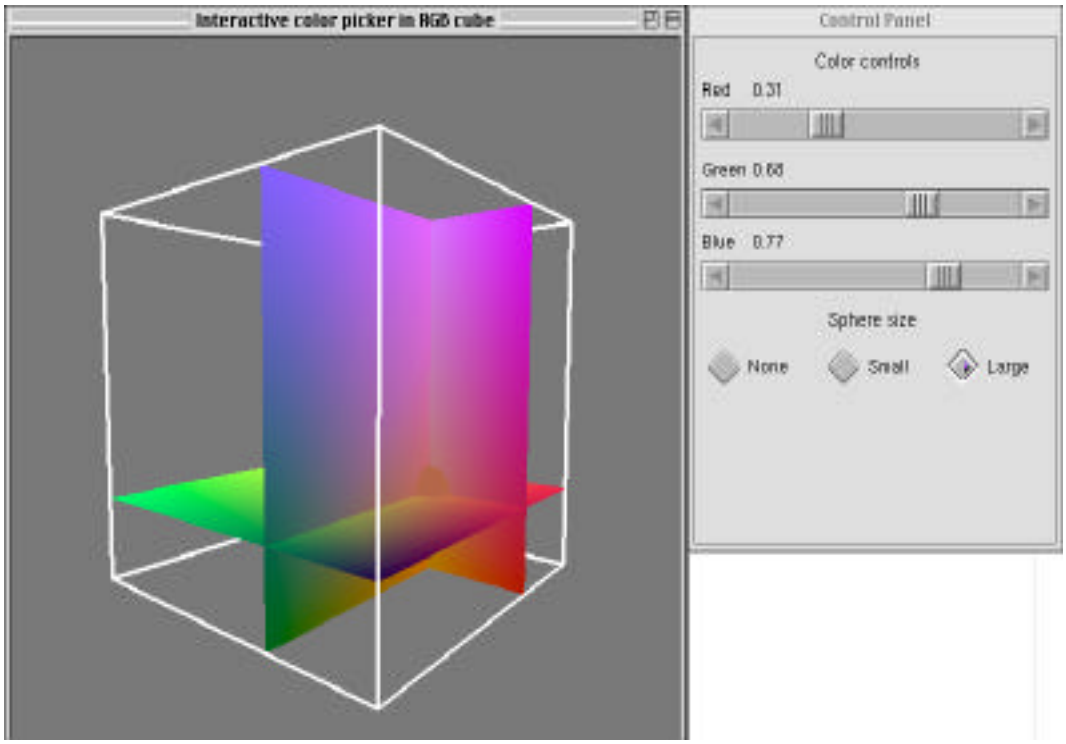


Figure 11.2: the color selector in context, with both the display and control windows shown

Installing MUI for Windows systems

MUI comes with the GLUT release, so if you have GLUT on your system you probably also have MUI. But if you do not have GLUT, when you download and uncompress the GLUT release you will have several header files (in the `include/mui` directory) and a couple of libraries: `libmui.a` for Unix and `mui.lib` for Windows. Install these in the usual places; for Windows, install `mui.lib` in the

```
<drive>:\Program Files\Microsoft Visual Studio\VC98\Lib\
```

directory. Place the header files also in the usual place; for Windows use

```
<drive>:\Program Files\Microsoft Visual Studio\VC98\include\
```

Then simply add `mui.lib` to your project files and you should be able to use MUI successfully.

A word to the wise...

The MUI control window has behaviors that are outside the programmer's control, so you must be aware of some of these in order to avoid some surprises. The primary behavior to watch for is that many of the MUI elements include a stream of events (and their associated redisplays) whenever the cursor is within the element's region of the window. If your application was not careful to

insulate itself against changes caused by redisplays, you may suddenly find the application window showing changes when you are not aware of requesting them or of creating any events at all. So if you use MUI, you should be particularly conscious of the structure of your application on redisplay and ensure that (for example) you clear any global variable that causes changes in your display before you leave the display function.

Chapter 11: Texture Mapping

Prerequisites

An understanding of the geometry of polygons in 3-space, a concept of interpolation across a polygon, and the concept of how values in one space can map linearly to values in another space.

Introduction

Texture mapping involves applying textures to a graphical object to achieve a more interesting image or add information to the image without computing additional geometry. This is a significant addition to the capabilities of computer graphics and is a very useful technique for you to master. Texturing is a rich topic and we will not try to cover it in all the depth that is possible, but we will describe the most useful set of capabilities as we develop the subject in a way that is compatible with current graphics APIs. This will allow you to use texture mapping effectively in your work.

The key idea of texture mapping is to apply additional information to your images as your geometry is computed and displayed. In the API environment we are assuming, the geometry is primarily based on polygons, and as the pixels of the polygon are computed, the color of each pixel is not calculated only from a simple lighting model but can include information from a texture map. This chapter will focus on how that texture mapping works. Most of the time we think of the texture as an image, so that when you render your objects they will be colored with the color values in the texture map. This approach allows you to use many tools to create visually-interesting things to be displayed on your objects. There are also ways to use texture maps to determine the luminance, intensity, or alpha values of your objects, adding significantly to the breadth of effects you can achieve.

Creating texture maps from images is not, however, the only approach that can be used in texture mapping. It is also possible to compute the texture data for each pixel of an object by procedural processes. This approach is more complex than we want to cover in depth in a first graphics programming course, but we will illustrate some procedural methods as we create texture maps for some of our examples. This will allow us to approximate procedural texturing and to give you an idea of the value of this kind of approach, and you can go on to look at these techniques yourself in more detail.

Texture maps are arrays of colors that represent information (for example, an image) that you want to display on an object in your scene. These maps can be 1D, 2D, or 3D arrays, though we will focus on 1D and 2D arrays here. Texture mapping is the process of identifying points on objects you define with points in a texture map to achieve images that can include strong visual interest while using simpler geometry.

The key point to be mastered is that you are dealing with two different spaces in texture mapping. The first is your modeling space, the space in which you define your objects to be displayed. The second is a space in which you create information that will be mapped to your objects. This information is in discrete pieces that correspond to cells in the texture array, often called *texels.* In order to use texture maps effectively, you must carefully consider how these two spaces will be linked when your image is created—you must include this relationship as part of your design for the final image.

In order to reconcile the two spaces used in texture mapping, and to have the information needed to develop the texture information to apply to each fragment, the system must be given values for

many individual parameters. You can think of this parameter setting as a binding operation, and it will be done by several API functions. Among the things you will need to set are

- the name (usually a small integer) given to the texture in internal texture memory,
- the dimensions of the texture map, the format of the information the texture map contains,
- what the texture map represents (a texture map may represent more than simply color),
- the way the texture and fragment information are to be combined when a fragment is rendered with the texture,
- how the texture is to be treated if the texture coordinates go outside the basic texture space,
- how the texture aliasing is to be handled when the texture is applied to each fragment, and
- whether the texture has a border.

You should look for all these bindings, and quite likely others, when you consider how your graphics API handles texture mapping.

There are many ways to create your texture maps. For 1D textures you may define a linear color function through various associations of color along a line segment. For 2D textures you may use scanned images, digital photos, digital paintings, or screen captures to create the images, and you may use image tools such as Photoshop to manipulate the images to achieve precisely the effects you want. Your graphics API may have tools that allow you to capture the contents of your frame buffer in an array where it can be read to a file or used as a texture map. This 2D texture world is the richest texture environment we will meet in these notes, and is the most common texture context for most graphics work. For 3D textures you may again define your texture by associating colors with points in space, but this is more difficult because there are few tools for scanning or painting 3D objects. However, you may compute the values of a 3D texture from a 3D model, and various kinds of medical scanning will produce 3D data, so 3D textures have many appropriate applications.

Most graphics APIs are quite flexible in accepting texture maps in many different formats. You can use one to four components for the texture map colors, and you can select RGB, RGBA, or any single one of these four components of color for the texture map. Many of these look like they have very specialized uses for unique effects, but an excellent general approach is to use straightforward 24-bit RGB color (8 bits per color per pixel) without any compression or special file formats—what Photoshop calls “raw RGB.”

Finally, texture mapping is much richer than simply applying colors to an object. Depending on the capabilities of your graphics API, you may be able to apply texture to a number of different kinds of properties, such as transparency or luminance. In the most sophisticated kinds of graphics, texturing is applied to such issues as the directions of normals to achieve special lighting effects such as bump mapping and anisotropic reflection.

Definitions

In defining texture maps below, we describe them as one-, two-, or three-dimensional arrays of colors. These are the correct definitions technically, but we usually conceptualize them a little more intuitively as one-, two-, or three-dimensional spaces that contain colors. When texture maps are applied, the vertices in the texture map may not correspond to the pixels you are filling in for the polygon, so the system must find a way to choose colors from the texture arrays. The graphics API provides ways to interpolate the vertices from the texture array and to compute the value for the pixel based on the colors at the interpolated vertices. These range from choosing the nearest point in the texture array to averaging the values of the colors for the pixel. However, this is usually not a problem when one first starts using textures, so we note this for future reference and will discuss how to do it for the OpenGL API later in this chapter.

1D texture maps: A 1D texture map is a one-dimensional array of colors that can be applied along any direction of an object, essentially as though it were extended to a 2D texture map by being

replicated into a 2D array. It thus allows you to apply textures that emphasize the direction you choose, and in our example below it allows us to apply a texture that varies only according to the distance of an object from the plane containing the eye point.

2D texture maps: A 2D texture map is a two-dimensional array of colors that can be applied to any 2D surface in a scene. This is probably the most natural and easy-to-understand kind of texture mapping, because it models the concept of "pasting" an image onto a surface. Another view of this could be that the image is on an elastic sheet and it is tacked onto the surface by pinning certain points of the sheet onto the vertices of the surface. By associating points on a polygon with points in the texture space, which are actually coordinates in the texture array, we allow the system to compute the association of any point on the polygon with a point in the texture space so the polygon point may be colored appropriately. When the polygon is drawn, then, the color from the texture space is used as directed in the texture map definition, as noted below.

3D texture maps: A 3D texture map is a three-dimensional array of colors. 3D textures are not supported in OpenGL 1.1, but were added in version 1.2. Because we assume that you will not yet have this advanced version of OpenGL, this is not covered here, but it is described in the OpenGL references [SHR] [WOO]. A useful visual examination of 3D textures is found in [WO00]. The 3D texture capability could be very useful in scientific work when the 3D texture is defined by an array of colors from data or theoretical work and the user can examine surfaces in 3-space, colored by the texture, to understand the information in the space.

Associating a vertex with a texture point

As you define your geometry, you associate a point in texture space with each vertex. This is similar to the way you associate a normal with each vertex when you set up lighting with smooth shading. This now can give you even more information associated with each point: the geometry of the coordinates, the color of the vertex or the normal for the vertex that allows the color to be computed, and the coordinates of the texture point that's associated with the vertex. The vertex information is applied to the image in the rendering pipeline step that processes fragments.

Depending on the way your graphics API works, you may either associate each vertex with actual texel coordinates of the texture point or a point with real coordinates, usually each in [0, 1], that represents a point by its proportional location in the texture map. The latter real-number approach is preferable because it allows your design for the use of textures to be independent of the actual texture map size.

The relation between the color of the object and the color of the texture map

In a texture-mapping situation, we have an object and a texture. The object may be assumed to have color properties, and the texture also has color properties. Defining the color or colors of the texture-mapped object involves considering the colors both the object and the texture map.

Perhaps the most common concept of texture mapping involves replacing any color on the original object by the color of the texture map. This is certainly one of the options that a graphics API will give you. But there are other options as well for many APIs. If the texture map has an alpha channel, you can blend the texture map onto the object, using the kind of color blending we discuss in the color chapter. You may also be able to apply other operations to the combination of object and texture color to achieve other effects. So don't assume simply that the only way to use texture maps is to replace the color of the object by the color of the texture; the options are much more interesting than merely that.

Other meanings for texture maps

Texture maps can describe other things besides an image that is to be mapped onto an object. A texture map can be used to change the appearance of a polygon by modifying the alpha value, luminance, or intensity of the pixels in the polygon based on the values in the texture map. This gives you a number of ways you can alter the appearance of a polygon by changing the way it is presented. This can be especially effective if it is used as part of multitexturing.

Creating texture maps

Any texture you use must be created somehow before it is loaded into the texture array. This may be done by using an image as your texture or by creating your texture through a computational process. In this section we will consider these two options and will outline how you can create a texture map through each.

Getting an image as a texture map

Using images as texture maps is very popular, especially when you want to give a naturalistic feel to a graphical object. Thus textures of sand, concrete, brick, grass, and ivy, to name only a few possible naturalistic textures, are often based on scanned or digital photographs of these materials. Other kinds of textures, such as flames or smoke, can be created with a digital paint system and used in your work. All the image-based textures are handled in the same way: the image is created and saved in a file with an appropriate format, and the file is read by the graphics program into a texture array to be used by the API's texture process. And we must note that at this time, all such textures are 2D textures because there are no generally-accepted formats for compressed 3D images.

The main problem with using images is that there is an enormous number of graphics file formats. Entire books are devoted to cataloging these formats [MUR], and some formats include compression techniques that require a great deal of computation when you re-create the image from the file. Using compressed images directly requires you to use a tool called an RIP—a raster image processor—to get the pixels from your image, and this would be a complex tool to write yourself. However, many implementations of graphics APIs are starting to include the ability to read images in various formats. Unless you have such functions available, we suggest that you avoid file formats such as JPEG, GIF, PICT, or even BMP and use only formats that store a simple sequence of RGB values. If you want to use an image that you have in a compressed file format, probably the simplest approach is to open the image in a highly-capable image manipulation tool such as Photoshop, which can read images in most formats, and then re-save it in a simplified form such as interlaced raw RGB.

A sample image that we will use as a texture map, a picture of a group of African penguins created from one of the author's photographs, is shown in Figure 11.1. Graphics APIs are likely to have restrictions on the dimensions of texture maps (for example, the OpenGL standard requires all dimensions, not including borders, to be a power of 2) so even if the format is so low-level that it does not include the dimensions, they can be recalled easily. We suggest that you include the dimension as part of the file name, such as `ivy.128x64.rgb` so that the size will not be something that must be recorded, and the process of using an image file as a texture map is described in the second code example in this chapter.

Figure 11.1: an image of that will be used as a texture map in several examples below

Generating a synthetic texture map

Because a texture map is simply an array of color, luminance, intensity, or alpha values, it is possible to generate the values of the array by applying a computational process instead of reading a file. Generating a texture computationally is a very powerful technique that can be very simple, or it may be relatively complex. Here we'll describe a few techniques that you might find helpful as a starting point in creating your own computed textures.

One of the simplest textures is the checkerboard tablecloth. For example, if we want to build a 64x64 texture array, we can define the color of an element `tex[i][j]` as red if `((i%4)+(j%4))%2` has value zero and white if the value is one. This will put a 4x4 red square at the top left of the texture and will alternate white and red 4x4 squares from there, thus creating a traditional checkerboard pattern. This kind of texture map is often used as a texture-mapped image because it shows problems easily, and such a texture map applied to two rectangles in space is shown in Figure 11.2.

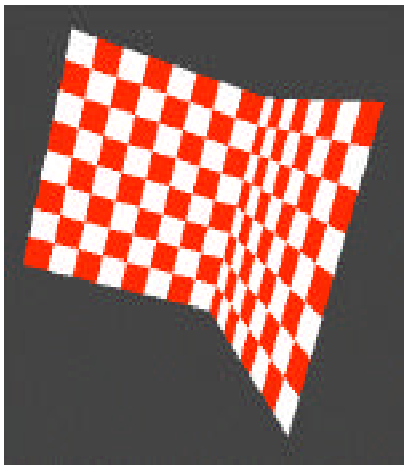


Figure 11.2: a simple checkered tablecloth pattern as a texture map

A particularly useful kind of computed texture involves using a *noise function.* A noise function is a single-valued function of one, two, or three variables that has no statistical correlation to any

rotation (that is, does not seem to vary systematically in any direction) or translation (does not seem to vary systematically across the domain) and that has a relatively limited amount of change in the value across a limited change in the domain. There are a number of ways to create such functions, and we will not begin to explore them all, but we will take one relatively simple approach to defining a noise function and use it to generate a couple of texture maps.

Instead of starting with a noise function itself, let's look at a simple texture map that has some of the properties of a noise function: no correlation for rotation or translation. If we generate a random number between 0 and 1 at each point of the texture map, then the nature of random numbers would give us the lack of correlation we need. But there is also no correlation between nearby elements of the texture map, so the purely random texture map is not satisfactory for many uses.

To give us a smoother, but still uncorrelated, random texture, we can apply the kind of filter function that we saw in the examples of diffusion processes in the science examples chapter. Recall that this filter replaces the value of each pixel with a weighted sum of the values of the pixels near that pixel, creating a weighted average of these values. If we start with a random texture and apply the filter process several times, we get a smoother texture that will still have the uncorrelated properties we want. This filtering can be done as often as you like, and the more often it is applied, the smoother the resulting texture. These processes are straightforward, and the results of 2D versions of the processes are shown in Figure 11.3. The texture can be created in grayscales, as shown, or colored textures can be created by creating similar textures for each of the RGB components in the color.

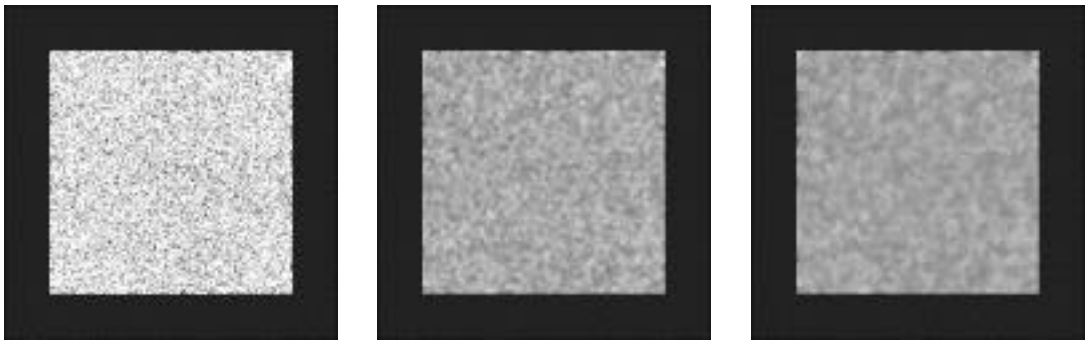


Figure 11.3: a random 2D texture (left) and the texture smoothed by filtering twice (center) and five times (right)

The random textures defined above are examples of noise functions that would be called *white noise*, with random values between 0 and 1. However, many other kinds of noise are possible, and one important one for texture maps is called 1/*f noise*. This is a noise function which is built from a linear combination of white noise functions f_N at various frequencies N, with the amplitude of f_N defined to be $1/N$. If we take these functions at frequencies 2^N for positive values of N, then the amplitude of the sum of the functions is $\Sigma(1/N)$, with the sum taken over all powers of two; this sum is 1, so the combined function also takes values from 0 to 1, just as the individual functions did. A texture map made from this technique has both large-scale properties (low frequencies) and small-scale details (high frequencies) and can be used to model some natural phenomena.

To see how this works, let's work our way through a simple 1D example. A noise function will be thought of as a piecewise linear function with values in [0,1], defined on the interval [0,1]. The *frequency* of a noise function on an interval can be defined to be the number of separate linear

segments the function has over the interval. So if a function is defined by values at 0.0, 0.5, and 1.0, its frequency is 2, while if the function is defined by values at 0,0. 0.25, 0.5, 0.75, and 1.0, its frequency is 4. It should be clear how to get functions with frequencies 8, 16, or any other power of 2: for frequency 2^N, the function f_N could be piecewise linear with values defined at $x=i/2^N$ for all values of i from 0 to 2^N. Figure 11.4 shows a very simple case (a piecewise linear function of one variable from 0 to 16, which could be used to create a 1D texture map 16 pixels wide) where the left-hand column is the graphs of the individual f_N functions for N = 2, 4, 8, and 16, the center column is the functions $f_N/2^N$, and the right-hand side is the sum of the functions in the center column. Obviously this is a very simple example and a more useful noise function—or noise texture map—would be defined over a larger interval and would involve more piecewise linear functions.

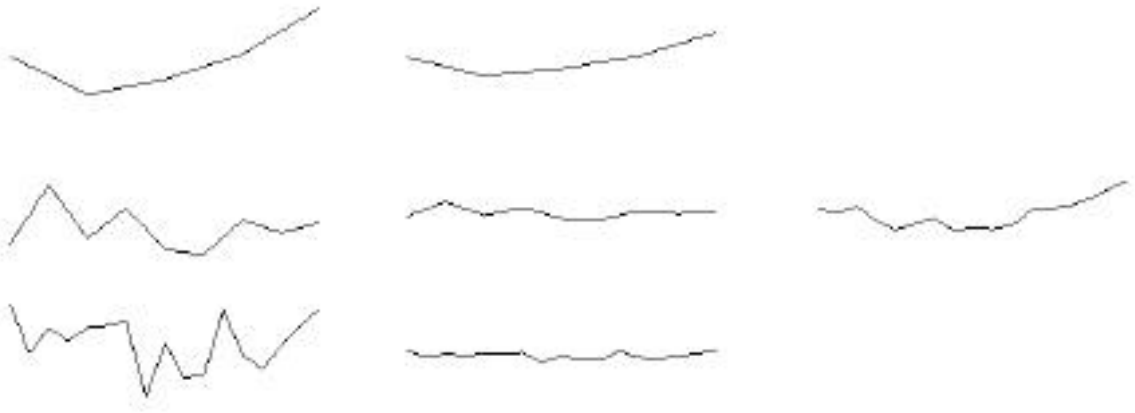


Figure 11.4: noise functions of various frequencies (left), multiplied by the reciprocals of their frequencies (center), and summed to create a single noise function (right).

In most cases, the function or texture map would use functions on 2D or 3D space instead of functions on 1D space. In the 2D case, you could use polygons defined by vertices with regular (x,y) coordinates and random z-coordinates, giving a set of points on a regular grid just as we described for graphing a 2D function in the chapter on science examples. In the 3D case you would need to compute values across a cubic region of space defined by eight coordinates in 3D space, which is much more difficult to visualize. But you can still work with functions of varying frequencies and sum them with varying weights, producing 1/*f* noise functions in 2D or 3D space.

In practice, the approach that is generally used is much more sophisticated than we have described. This approach uses gradient interpolation as discussed by Peachy in [Ebert]. This is also the kind of noise function used in the Renderman™ shader system. We will not describe this in detail, but you are encouraged to work through the sample code for such noise functions provided by Mike Bailey that is included in the supplementary materials for this book. Figure 11.5 is an example of a texture built with such a process. As an overview, let's consider this in the 3D noise case.

The general process is to start with a 3D mesh of the right frequency for a particular function f_N. We will treat the coordinates of each 3D mesh point as *x*-, *y*-, and *z*-components of a point in the noise function domain, and the process computes a unit vector of three random components that represents the gradient at that point. These three components are the direction and amplitude of the gradient. We will then assume a height of 0 at each grid point and use the gradients to define a

smooth function to be our basic noise function. You are encouraged to read the discussions in [Ebert] to understand this process and some of the issues in using it effectively and efficiently.

Figure 11.5: a texture from a 1/f noise function

Texture mapping and billboards

In the chapter on high-performance graphics techniques we introduce the concept of a *billboard*—a two-dimensional polygon in three-dimensional space that is always rotated to face the viewer and that has an image texture-mapped onto it so that the image on the polygon seems to be a three-dimensional object in the scene. This is a straightforward application of texture mapping but requires that the color of the polygon come entirely from the texture map and that some portions of the texture map have a zero alpha value so they will seem transparent when the polygon is displayed.

Interpolation for texture maps

A 2D texture-mapped polygon is created by interpolating the texture coordinates for each pixel in the polygon from the texture coordinates of the polygon vertices in the rendering pipeline. As we noted in the chapter on that pipeline, if the scene uses a perspective projection, the highest quality texturing is done if the interpolation takes the perspective into account by back-projecting the 2D coordinates for each pixel into the original modeling space before doing the interpolation. If this perspective correction is not done, the texture in each polygon follows a linear pattern based on the polygon boundaries, which can cause awkward artifacts at polygon boundaries. In Figure 11.6, we see a quad defined as two triangles with a checkerboard texture, and we see that in the triangle at lower left, all the lines in the texture are parallel to the left or bottom edges while all the lines in

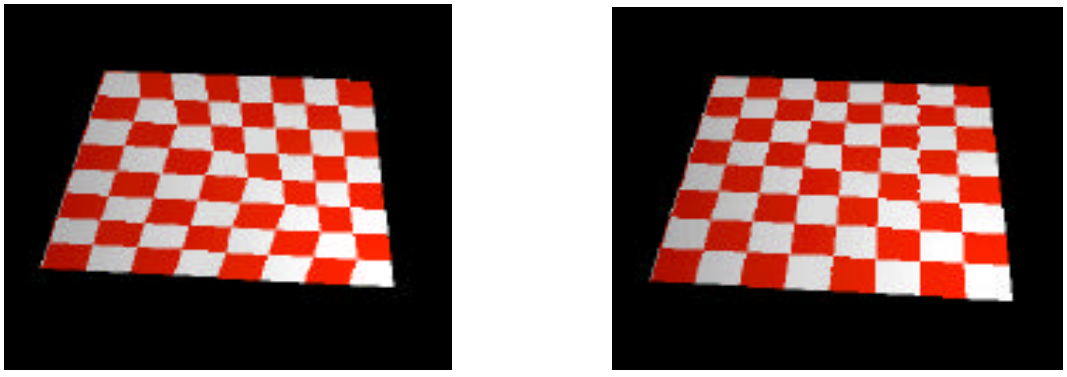


Figure 11.6: a planar rectangular region defined by two triangles without (left) and with (right) perspective correction for the texture mapping

the upper right triangle are parallel to the right or top edges. But because the quad is shown with a perspective projection, the left and right sides are not parallel, leading to problems with the texture where the triangles meet. This illustrates the difference that perspective-corrected interpolation can make for texture mapping. Similar problems can occur for 1D or 3D texture maps in a perspective projection situation when perspective correction is not used. This figure also shows how readily the checkerboard texture shows problems in textured images, as we suggested earlier in the chapter.

The technique for doing perspective-corrected texture interpolation is simply a special case of the perspective-corrected interpolation discussed in the chapter on the graphics pipeline. Of course, this problem will not occur if you use an orthogonal projection because with this projection, the linear relation between pixels in the screen space is exactly mirrored by the linear relation between the original points in model space.

Antialiasing in texturing

When you apply a texture map to a polygon, you identify the vertices in the polygon with values in texture space. These values may or may not be integers (that is, actual indices in the texture map) but the interpolation process we discussed will assign a value in texture space to each pixel in the polygon. The pixel may represent only part of a texel (texture cell) if the difference between the texture-space values for adjacent pixels is less than one, or it may represent many texels if the difference between the texture space values for adjacent pixels is greater than one. This offers two kinds of aliasing—the magnification of texels if the texture is coarse relative to the object being texture mapped (a magnification filter), or the selection of color from widely separated texels if the texture is very fine relative to the object (a minification filter).

Because textures may involve aliasing, it can be useful to have antialiasing techniques with texturing. For magnification filtering, you will find yourself with pixel coordinates often having two adjacent points within the same texel. You can choose to use the *nearest* filter to determine the color of a pixel: the color is set to the color of the nearest texel vertex. This can alias a number of pixels to the color of a single texel vertex, and can give you a blocky image. Another approach is to use linear filtering, where each pixel's color is determined by a weighted average of the texel vertices around it, with the weight being determined by how close the pixel is to each texel vertex. Other, more sophisticated kinds of antialiasing techniques are also possible, but graphics APIs tend to keep things simple. In the OpenGL API, the only antialiasing tool available is the linear filtering that we discuss below but other APIs may have other tools, and certainly sophisticated, custom-built or research graphics systems can use a full set of antialiasing techniques. This needs to be considered when considering the nature of your application and choosing your API. See [Ebert] for more details.

MIP Mapping

We saw that when there is only a single texture map available, the graphics system must sometimes use some sort of antialiasing process to choose the color of a pixel from the colors of pixels in the original map. If the pixel space of a polygon is larger than that of the texture map, there is no way to get individual texel information for each pixel and techniques such as linear filtering are needed. But as a polygon gets small, the pixel space gets smaller than the texture space and you will find that pixels that are near each other in the polygon have colors that are not near each other in the texture space. As these polygons move, the colors can jump around unpredictably, causing unwanted effects.

A solution to this problem can be provided by giving your system a hierarchy of texture maps of different sizes, and having the system select the map that best fits the size of your polygon. One

technique for doing this is MIP mapping (MIP means *multum in parvo*, or "many things in a small place.") With this approach, you provide your texture map in many resolutions so you can control the versions of the texture that will be seen at each level. This set of different resolution maps is all held the same texture memory and the proper one is selected depending on the size of the polygon to be presented.

MIP mapping can be seen as a level-of-detail process (see the chapter on high-performance graphics) but it is used less for performance reasons than for quality reasons. Thus we believe it fits best in the discussion here of providing quality texture mapping.

Multitexturing

Multitexturing is a rendering technique in which two or more textures are applied to a single surface in the rendering process, applying more than one texture environment to the model as shown in Figure 11.7. For example, one texture might be a wood surface and a second texture might be a light map. The combination of the two textures would produce a texture of a lighted wooden surface. This use of surface and light maps is a common technique in games programming. Other examples might include combining aerial photographs, GIS (geographic information systems) locator symbols, and elevation contour lines to produce a map that combines realistic terrain, points of interest, and elevation information.

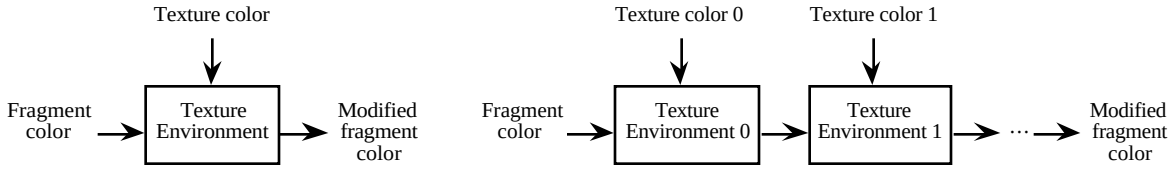


Figure 11.7: texturing in the graphics pipeline: single texture (left) and multitexture (right)

With standard texturing capabilities you might be able to achieve similar results. It may be possible to combine separate texture maps into a single texture map if the separate maps are of the same size and share the same texture coordinates on the target to be textured. You would simply read the values in the individual texture arrays and combine them with appropriate operations into a new texture array that you could use as your single texture. However, this is very restrictive, because texture data usually does not come in such nice packages, and separate texture maps may be of different sizes and with different orientations relative to the surface to be textured. So having a multitexture capability allows you to use each texture with its own individual properties, and the combination you need will be applied in the rendering process.

At this point, multitexturing is not part of many graphics API standards, but it is common in some development systems and can be expected to become part of either API standards or API extensions. This is primarily driven by two forces: graphics programmers who want to user more sophisticated texturing than is available with current standards, and the fact that graphics boards are now coming out with multitexturing capabilities.

In general, using multitextures is almost exactly like using several individual textures. You need to create individual texture maps from whatever sources you need, and you need to specify that you will be using a number of textures and bind the particular texture maps to each and enable them. When you specify your geometry, you will need to define the texture coordinate for each of the individual textures that corresponds to each vertex point. This is not especially difficult, and in the case of the OpenGL multitexturing extension, this is discussed later in the chapter.

As an example of the kind of result that multitexturing can give you, Figure 11.8 (due to Bryan McNett; permission has not yet been sought) shows a surface texture and a light map separately and the results when the textures are applied together to a polygon.

Figure 11.8: showing multitexturing in use

Using billboards

Because we mentioned billboards in this chapter, it seems like a good time to describe how to build them and use them in your programs. The reason for using a billboard is to show the viewer an image of a 3D object with 2D techniques, giving the illusion of 3D actions in the scene.

The key point for a billboard is that it must point directly toward the viewer at all times. In order to accomplish this, we consider the scene graph for the scene in question; there will be a set of transformations that set the position orientation of a rectangle that was initially placed in the scene. You must take these transformations and invert them, as discussed in the chapter on modeling with scene graphs, giving a transformation that would place them at the root of the scene graph. Then you must add any transformations that are used to orient the eye point. This transformation, then, is the final modeling transformation defined before the billboard is drawn, and will always make the billboard point toward the viewer.

Texture mapping in OpenGL

There are many details to master before you can count yourself fully skilled at using textures in your images. The full details must be left to the manuals for OpenGL or another API, but here we will discuss many of them, certainly enough to give you a good range of skills in the subject. The details we will discuss are the texture environment, texture parameters, building a texture array, defining a texture map, and generating textures. We will have examples of many of these details to help you see how they work.

One of the details you need to understand is which texture-related functions are used to define which aspects of texture mapping. While there are not a large number of these functions, you must use them carefully and in an appropriate sequence in order to have your textures work properly. A list of the primary texture-related functions in OpenGL is given here, and later in the chapter you will see more details of their use.

- `glEnable(…)` enable texture mapping as needed in your program; also `glDisable(…)` to disable texture mapping when no longer needed
- `glGenTextures(…)` generates one or more names (integers) that can be used for textures
- `glBindTexture(…)` binds a texture name (generated in glGenTextures) to a texture target (`GL_TEXTURE_1D`,`GL_TEXTURE_2D`,`GL_TEXTURE_3D`)
- `glTexImage*(…)` binds values to most of the parameters that are used to define a texture, such as the number of color coordinates and the internal

format of the texture data, how the texture is to be interpreted, the size of the texture map, and the like.

- `glTexEnv*(…)` defines the action of the texture when it is applied to the object on a per-fragment basis
- `glTexCoord*(…)` associates a texture coordinate to a vertex of a graphic object
- `glTexParameter*(…)` defines how antialiasing, wrapping, and similar functions are to be applied to the texture
- `glTexGen*(…)` controls the automatic generation of vertex coordinates for an object
- `glDeleteTextures(…)` deletes one or more textures that had been generated by glGenTextures

Associating vertices and texture points

You define your geometry in OpenGL by using the basic primitives as we described earlier in these notes. Within a `glBegin(…)` … `glEnd()` pair, you are used to including both `glVertex*()` and `glNormal*()` functions; you can also include `glTexCoord*()` functions to define the texture coordinate for each vertex. As always, these functions come in several varieties, depending on what kind of coordinates you use, including

```
glTexCoord1f(float)
glTexCoord2f(float, float)
glTexCoord3f(float, float, float)
glTexCoord1fv(float[1])
glTexCoord2fv(float[2])
glTexCoord3fv(float[3])
```

When you specify the texture coordinates, you must do so before you give the `glVertex()` function that defines the point because the state of vertex is set at that point.

The actual texture coordinates represent the real number that represents the position of the texture point within the texture map, so coordinates in the range [0, 1] give you points within the texture space. Coordinates outside this range are interpreted according to your choice of texture wrap or clamp, as described later.

Capturing a texture from the screen

A useful approach to textures is to create an image and save the color buffer (the frame buffer) as an array that can be used as a texture map. This can allow you to create a number of different kinds of images for texture maps. This operation is supported by many graphics APIs. For example, in OpenGL, the `glReadBuffer(mode)` function determines the color buffer from which subsequent buffer reads are to be done, usually the front buffer if you are using single buffering or the back buffer if you are using double buffering. The `glReadPixels(...)` function, used with the RGB or RGBA symbolic format, can then copy the values of the elements in that buffer into a target array. This function can do much more, however; it can save the values of any one color channel, of the depth buffer, or of the luminance, among others. As such it gives you the ability to retrieve a number of different kinds of information from an image. We will not go into more detail here but refer the student to the manuals for the use of these functions.

The array returned by the `glReadPixels(...)` function may be written to a file for later use, or may be used immediately in the program as the texture array. If it is saved to a file, it will probably be most useful if it is saved in a very raw format, holding nothing but the values read from the buffer, but you may want to add extra information to allow it to be used more readily. For example, if you start the file with the width and height of the image, your file will resemble the .ppm format that can be used by many image manipulation programs. If you capture a stream of images into files with names that include sequential numbers, it may be possible to write scripts

that will pick these up and make them into a digital movie to display your images as an animation. We refer you to the chapter on animation for more details.

Texture environment

The a graphics API, you must define your texture environment to specify how texture values are to be used when the texture is applied to a polygon. In OpenGL, the appropriate function call is

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, *)
```

The meaning of the texture is determined by the value of the last parameter. The options are `GL_BLEND`, `GL_DECAL`, `GL_MODULATE`, or `GL_REPLACE`.

If the texture represents *RGB color*, the behavior of the texture when it is applied is defined as follows. In this and the other behavior descriptions, we use *C*, *A*, *I*, and *L* for color, alpha, intensity, and luminance respectively, and subscripts f and t for the fragment and texture values.

`GL_BLEND`: the color of the pixel is $C_f(1\text{-}C_t)$.

`GL_DECAL`: the color of the pixel is C_t, simply replacing the color by the texture color.

`GL_MODULATE`: the color of the pixel is $C_f{*}C_t$, replacing the color by the subtractive computation for color.

`GL_REPLACE`: same as GL_DECAL for color.

If the texture represents *RGBA color*, then the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1\text{-}C_t)$, and the alpha channel in the pixel is $A_f{*}A_t$.

`GL_DECAL`: the color of the pixel is $(1\text{–}A_t)C_f{+}A_tC_t$, and the alpha channel in the pixel is A_f.

`GL_MODULATE`: the color of the pixel is $C_f{*}C_t$, as above, and the alpha channel in the pixel is $A_f{*}A_t$.

`GL_REPLACE`: the color of the pixel is C_t and the alpha channel in the pixel is A_t.

If the texture represents the *alpha channel*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is C_f, and the alpha channel in the pixel is A_f.

`GL_DECAL`: the operation is undefined

`GL_MODULATE`: the color of the pixel is C_f, and the alpha channel in the pixel is $A_f{*}A_t$.

`GL_REPLACE`: the color of the pixel is C_f and the alpha channel in the pixel is A_t.

If the texture represents *luminance*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1\text{-}L_t)$, and the alpha channel in the pixel is A_f.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is $C_f{*}L_t$, and the alpha channel in the pixel is A_f.

`GL_REPLACE`: the color of the pixel is L_t and the alpha channel in the pixel is A_f.

If the texture represents *intensity*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1\text{-}I_t)$, and the alpha channel in the pixel is $A_f(1\text{-}I_t)$.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is $C_f{*}I_t$, and the alpha channel in the pixel is $A_f{*}I_t$.

`GL_REPLACE`: the color of the pixel is I_t and the alpha channel in the pixel is I_t.

Texture parameters

The texture parameters define how the texture will be presented on a polygon in your scene. In OpenGL, the parameters you will want to understand include texture wrap and texture filtering. Texture wrap behavior, defined by the `GL_TEXTURE_WRAP_*` parameter, specifies the system behavior when you define texture coordinates outside the [0,1] range in any of the texture dimensions. The two options you have available are repeating or clamping the texture. Repeating the texture is accomplished by taking only the decimal part of any texture coordinate, so after you go beyond 1 you start over at 0. This repeats the texture across the polygon to fill out the texture space you have defined. Clamping the texture involves taking any texture coordinate outside [0,1] and translating it to the nearer of 0 or 1. This continues the color of the texture border outside the region where the texture coordinates are within [0,1]. This uses the `glTexParameter*(...)` function to repeat, or clamp, the texture respectively as follows:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

Using these parameters, which define horizontal clamping and vertical repeating, produces an image like that of Figure 11.9.

Figure 11.9 a quad with a texture that is wrapped in one direction and clamped in the other

If you will be using repeating textures you will effectively be tiling your polygons, and it worth thinking a moment about what makes a good tiling texture. A tiling figure needs to have the same colors and overall textures at both the left and right hand side of the figure as well as at the top and bottom side. This is typically not easy to find, so there are techniques to make good tiling figures. One is to use a tool such as Photoshop and rotate the figure so that the former edges of the figure are in the middle of the new image and are adjacent to each other. Using the Photoshop tools, the middle of the figure is blurred or manipulated so that the line formerly in the middle is not visible. When the picture is then rotated back so that the edges are back where they started, its left and right sides will tile correctly. A similar operation applied to the top and bottom figures completes the process of making the tile figure.

Another important texture parameter controls the filtering for pixels to deal with aliasing issues. In OpenGL, this is called the minification (if there are many texture points that correspond to one pixel in the image) or magnification (if there are many pixels that correspond to one point in the texture) filter, and it controls the way an individual pixel is colored based on the texture map. For any pixel in your scene, the texture coordinate for the pixel is computed through an interpolation across a polygon, and rarely corresponds exactly to an index in the texture array, so the system must create the color for the pixel by a computation in the texture space. You control this in

OpenGL with the texture parameter GL_TEXTURE_*_FILTER that you set in the glTexParameter*(...) function. The filter you use depends on whether a pixel in your image maps to a space larger or smaller than one texture element. If a pixel is smaller than a texture element, then GL_TEXTURE_MIN_FILTER is used; if a pixel is larger than a texture element, then GL_TEXTURE_MAG_FILTER is used. An example of the usage is:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

The symbolic values for these filters are GL_NEAREST or GL_LINEAR. If you choose the value GL_NEAREST for the filter, then the system chooses the single point in the texture space nearest the computed texture coordinate; if you choose GL_LINEAR then the system averages the four nearest points to the computed texture coordinate with weights depending on the distance to each point. The former is a faster approach, but has problems with aliasing; the latter is slower but produces a much smoother image. This difference is illustrated in Figure 11.10 in an extreme close-up of the penguin image, and it is easy to see that choosing GL_NEAREST for the mag filter gives a much coarser image than the GL_LINEAR filter. Your choice will depend on the relative importance of speed and image quality in your work.

Figure 11.10: the penguin head texture with the GL_NEAREST (left) and GL_LINEAR (right) magnification filters

Getting and defining a texture map

This set of definitions is managed by the glTexImage*D(...) functions. These are a complex set of functions with a number of different parameters. The functions cover 1D, 2D, and 3D textures (the dimension is the asterix in the function name) and have the same structure for their parameters.

Before you can apply the glTexImage*D(...) function, however, you must define and fill an array that holds your texture data. This array of unsigned integers (GLuint) will have the same dimension as your texture. The data in the array can be organized in many ways, as we will see when we talk about the internal format of the texture data. You may read the values of the array from a file or you may generate the values through your own programming. The examples in this chapter illustrate both options.

The `glTexImage*D(...)` function has one of the more complex parameter lists. These parameters are, in order,

- the *target*, usually `GL_TEXTURE_*D`, where * is 1, 2, or 3. Proxy textures are also possible, but are beyond the range of topics we will cover here. This target will be used in a number of places in defining texture maps.
- the *level*, an integer representing level-of-detail number. This supports multiple-level MIP-mapping.
- the *internal format* of the texture map, one of the places where an API such as OpenGL must support a large number of options to meet the needs of a wide community. For OpenGL, this internal format is a symbolic constant and can vary quite widely, but we will list only a set we believe will be most useful to the student. Most of the other options deal with other organizations that involve a different number of bits per pixel of the component. Here we deal only with formats that have eight bits per component, and we leave the others (and information on them in manuals) to applications that need specialized formats.
  - `GL_ALPHA8`
  - `GL_LUMINANCE8`
  - `GL_INTENSITY8`
  - `GL_RGB8`
  - `GL_RGBA8`
- the *dimensions* of the texture map, of type `GLsizei`, so the number of parameters here is the dimension of the texture map. If you have a 1D texture map, this parameter is the *width*; if you have a 2D texture map, the two parameters are the width and *height*; if you have a 3D texture map, the three parameters are width, height, and *depth*. Each of these must have a value of 2^N+2*(*border*) for some integer *N*, where the value of *border* is either 0 or 1 as specified in the next parameter.
- the *border*, an integer that is either 0 (if no border is present) or 1 (if there is a border).
- the *format*, a symbolic constant that defines what the data type of the pixel data in the texture array is. This includes the following, as well as some other types that are more exotic:
  - `GL_ALPHA`
  - `GL_RGB`
  - `GL_RGBA`
  - `GL_LUMINANCE`

  The format indicates how the texture is to be used in creating the image. We discussed the effects of the texture modes and the texture format in the discussion of image modes above.
- the *type* of the pixel data, a symbolic constant that indicates the data type stored in the texture array per pixel. This is usually pretty simple, as shown in the examples below which use only `GL_FLOAT` and `GL_UNSIGNED_BYTE` types.
- the *pixels*, an address of the pixel data (texture array) in memory.

So the complete function call is

```
glTexImage*D(target, level, internal format, dimensions,
             border, format, type, pixels)
```

An example of this complete function call can be found below for the 2D texture on the surface of a cube.

Note that the `glTexImage*D(...)` function simply defines how the texture array is stored and what it is taken to mean. It does not say anything about the source of the image; if you want to use a compressed image format to store your image outside the file, it would have to be translated in order to put the content into the *pixels* array.

You will be creating your textures from some set of sources and probably using the same kind of tools. When you find a particular approach that works for you, you'll most likely settle on that particular approach to textures. The number of options in structuring your texture is phenomenal,

as you can tell from the number of options in some of the parameters above, but you should not be daunted by this broad set of possibilities and should focus on finding an approach you can use.

Texture coordinate control

As your texture is applied to a polygon, you may specify how the texture coordinates correspond to the vertices with the `glTexture*(...)` function, as we have generally assumed above, or you may direct the OpenGL system to assign the texture coordinates for you. This is done with the `glTexGen*(...)` function, which allows you to specify the details of the texture generation operation.

The `glTexGen*(...)` function takes three parameters. The first is the texture coordinate being defined, which is one of `GL_S`, `GL_T`, `GL_R`, or `GL_Q` with S, T, R, and Q being the first, second, third, and homogeneous coordinates of the texture. The second parameter is one of three symbolic constants: `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. If the second parameter is `GL_TEXTURE_GEN_MODE`, the third parameter is a single symbolic constant with value `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, or `GL_SPHERE_MAP`. If the second parameter is `GL_OBJECT_PLANE`, the third parameter is a vector of four values that defines the plane from which an object-linear texture is defined; if the second parameter is `GL_EYE_PLANE`, the third parameter is a vector of four values that defines the plane that contains the eye point. In both these cases, the object-linear or eye-linear value is computed based on the coefficients. If the second parameter is `GL_TEXTURE_GEN_MODE` and the third parameter is `GL_SPHERE_MAP`, the texture is generated based on an approximation of the reflection vector from the surface to the texture map.

Applications of this texture generation include the Chromadepth™ texture, which is a 1D eye-linear texture generated with parameters that define the starting and ending points of the texture. Another example is automatic contour generation, where you use a `GL_OBJECT_LINEAR` mode and the `GL_OBJECT_PLANE` operation that defines the base plane from which contours are to be generated. Because contours are typically generated from a sea-level plane (one of the coordinates is 0), it is easy to define the coefficients for the object plane base.

Texture interpolation

As we noted earlier in the chapter, the scanline interpolation done to render a polynomial needs to take perspective into account to get the highest possible texture quality if the image projection is perspective. In the rasterization step of the rendering pipeline, both the endpoints of each scanline and the internal pixels of the polygon itself are filled by an interpolation process based on the vertex coordinates. For each scan line of the frame buffer that meets the polygon, the endpoints of the scan are interpolated from appropriate vertices and each pixel between them has its color, depth, and optionally texture coordinates computed to create what is called a fragment. These interpolated points are affected by the quality you specify with the OpenGL hint function

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, hint).
```

Here the hint may be `GL_DONT_CARE` (take the system default), `GL_NICEST` (perform the perspective correction to get the best image), or `GL_FASTEST` (don't perform the perspective correction to maximize speed). These fragments are then passed to the per-fragment operations

Texture mapping and GLU quadrics

As we noted in the chapter on modeling, the GLU quadric objects have built-in texture mapping capabilities, and this is one of the features that makes them very attractive to use for modeling. To use these, we must carry out three tasks: load the texture to the system and bind it to a name,

define the quadric to have normals and a texture, and then bind the texture to the object geometry as the object is drawn. The short code fragments for these three tasks are given below, with a generic function `readTextureFile(...)` specified that you will probably need to write for yourself, and with a generic GLU function to identify the quadric to be drawn.

```
readTextureFile(...);
glBindTexture(GL_TEXTURE_2D, texture[i]);
glTexImage2D(GL_TEXTURE_2D,...);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

myQuadric = gluNewQuadric();
gluQuadricNormals(myQuadric, GL_SMOOTH);
gluQuadricTexture(myQuadric, GL_TRUE);
gluQuadricDrawStyle(myQuadric, GLU_FILL);

glPushMatrix();
   // modeling transformations as needed
   glBindTexture(GL_TEXTURE_2D, texture[i]);
   gluXXX(myQuadric, ...);
glPopMatrix();
```

Multitextures

Multitexturing is an optional part of OpenGL 1.2 but not of earlier versions of the OpenGL API. The ARB_multitexture extension, approved by the OpenGL Architecture Review Board, is a relatively common OpenGL extension for earlier versions of the API. However, the extension is currently deprecated in favor of version 1.2 when it includes multitexturing.

Multitexuring operates by defining multiple texture objects with the OpenGL function `glGenTextures(N, texNames)`. There is no guaranteed minimum number of textures that are supported, but you may inquire that number of your system. For each of the texture objects, you define the properties of the texture through the functions `glTexImage*()` and `glTexParameteri()` in the same way you would for a single texture. You then define texture units for each of your textures with the `glBindTexture()` and `glTexEnvi()` functions, giving you a set of textures that will be applied in the order of their name indices. When an object is rendered with these textures, the first texture will be applied first, the second texture to the object that is the result of the first texture mapping, and so on.

In actually applying the textures to an object, you must assign the texture coordinates for each of your textures to the each of the vertices of the object. This is illustrated in the multitexturing code example later in this chapter.

Some examples

Textures can be applied in several different ways with the function

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode )
```

One way uses a decal technique, with mode `GL_DECAL`, in which the content of the texture is applied as an opaque image on the surface of the polygon, showing nothing but the texture map. Another way uses a modulation technique, with mode `GL_MODULATE`, in which the content of the texture is displayed on the surface as though it were colored plastic. This mode allows you to show the shading of a lighted surface by defining a white surface and letting the shading show through the modulated texture. There is also a mode `GL_BLEND` that blends the color of the object with the color of the texture map based on the alpha values, just as other color blending is done. In

the examples below, the Chromadepth image is created with a 1D modulated texture so that the underlying surface shading is displayed, while the mapped-cube image is created with a 2D decal texture so that the face of the cube is precisely the texture map. You may use several different textures with one image, so that (for example) you could take a purely geometric white terrain model, apply a 2D texture map of an aerial photograph of the terrain with `GL_MODULATE` mode to get a realistic image of the terrain, and then apply a 1D texture map in `GL_BLEND` mode that is mostly transparent but has colors at specific levels and that is oriented to the vertical in the 3D image in order to get elevation lines on the terrain. Your only limitation is your imagination — and the time to develop all the techniques.

The Chromadepth™ process: using 1D texture maps to create the illusion of depth. If you apply a lighting model with white light to a white object, you get a pure expression of shading on the object. If you then apply a 1D texture by attaching a point near the eye to the red end of the ramp (see the code below in the 1D color ramp example) and a point far from the eye to the blue end of the ramp, you get a result like that shown in Figure 11.11 below.

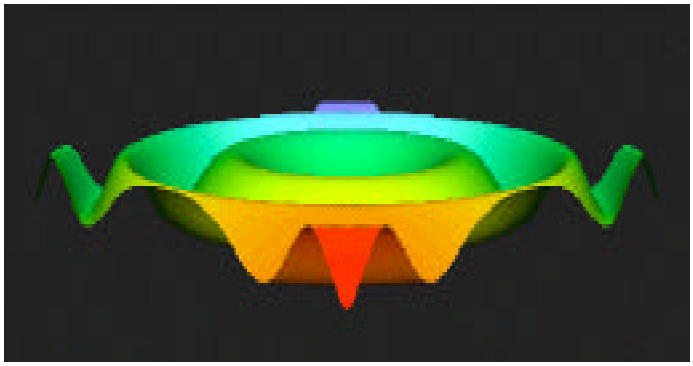


Figure 11.11: a Chromadepth-colored image of a mathematical surface

This creates a very convincing 3D image when it is viewed through Chromadepth™ glasses, because these glasses have a diffraction grating in one lens and clear plastic in the other. The diffraction grating bends red light more than blue light, so the angle between red objects as seen by both eyes is larger than the angle between blue objects. Our visual system interprets objects having larger angles between them as closer than objects having smaller angles, so with these glasses, red objects are interpreted as being closer than blue objects.

Figure 11.12: a 3D cube with the penguin texture map on one face

Using 2D texture maps to add interest to a surface: often we want to create relatively simple objects but have them look complex, particularly when we are trying to create models that mimic things in the real world. We can accomplish this by mapping images (for example, images of the real world) onto our simpler objects. In the very simple example shown in Figure 11.12, the penguin image was used as the texture map on one face of a cube. This texture could also have been created by saving the frame buffer into a file in the program that created the texture map. This created a cube that has more visual content than its geometry would suggest, and it was extremely simple to connect the square image with the square face of the cube.

Sample code for this is in three parts. The first reads a file into a texture array; the second sets up the OpenGL functions that define how the texture map is to be applied; and the third draws the face of the cube with the texture map applied.

```
void setTexture(void) // read file into RGB8 format array
{
   FILE * fd;
   GLubyte ch;
   int i,j,k;

   fd = fopen("penguin.512.512.rgb", "r");
   for (i=0; i<TEX_WIDTH; i++) {
      for (j=0; j<TEX_HEIGHT; j++) {
         for (k=0; k<3; k++) {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
         }
      }
   }
   fclose(fd);
}

// enable textures for the last face
   glEnable(GL_TEXTURE_2D);
   glGenTextures(1, texName); // define texture for sixth face
   glBindTexture(GL_TEXTURE_2D,texName[0]);
   glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
   glTexImage2D(GL_TEXTURE_2D,0,GL_RGB8,TEX_WIDTH,TEX_HEIGHT,
                0,GL_RGB,GL_UNSIGNED_BYTE,texImage);
   glBindTexture(GL_TEXTURE_2D, texName[0]);

   glBegin(GL_QUADS);          // sixth quad: negative X face
      glNormal3fv(normals[1]);  // single normal; flat shading
      glTexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
      glTexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
      glTexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
      glTexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
   glEnd();
   glDeleteTextures(1, texName);
```

Environment maps

Environment maps allow us to create the illusion that an object reflects images from a texture that we define. This can provide some very interesting effects, because realistic reflections of real-

world objects is one of the visual realism clues we would expect. With environment maps, we can use photographs or synthetic images as the things we want to reflect, and we can adapt the parameters of the texture map to give us realistic effects. One of the easy effects to get is the reflection of things in a chrome-like surface. In Figure 11.13, we see an example of this as a photograph of Hong Kong that has been modified in Photoshop with a spherical filter is used as a texture map on a surface. The lens effect makes the environment map much more convincing because the environment map uses the surface normals at a point to identify the texture points for the final image.

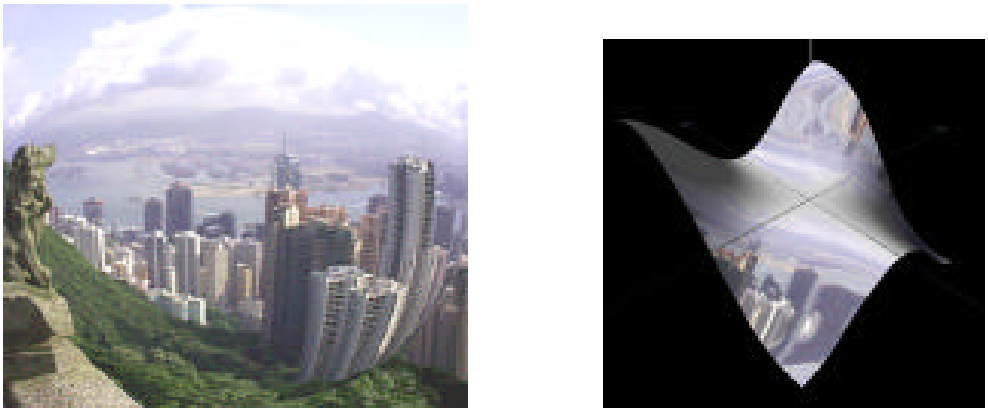


Figure 11.13: the original texture for an environment map (left) and the map on a surface (right)

A word to the wise...

Texture mapping is a much richer subject than these fairly simple examples have been able to show. You can use 1D textures to provide contour lines on a surface or to give you the kind of color encoding for a height value we discussed in the module on visual communication. You can use 2D textures in several sophisticated ways to give you the illusion of bumpy surfaces (use a texture on the luminance), to give the effect of looking through a varigated cloud (use a fractal texture on alpha) or of such a cloud on shadows (use the same kind of texture on luminance on a landscape image). This subject is a fruitful area for creative work.

There are several points that you must consider in order to avoid problems when you use texture mapping in your work. If you select your texture coordinates carelessly, you can create effects you might not expect because the geometry of your objects does not match the geometry of your texture map. One particular case of this is if you use a texture map that has a different aspect ratio than the space you are mapping it onto, which can change proportions in the texture that you might not have expected. More serious, perhaps, is trying to map an entire rectangular area into a quadrilateral that isn't rectangular, so that the texture is distorted nonlinearly. Imagine the effect if you were to try to map a brick texture into a non-convex polygon, for example. Another problem can arise if you texture-map two adjacent polygons with maps that do not align at the seam between the polygons. Much like wallpaper that doesn't match at a corner, the effect can be disturbing and can ruin any attempt at creating realism in your image. Finally, if you use texture maps whose resolution is significantly different from the resolution of the polygon using the texture, you can run into problems of aliasing textures caused by selecting only portions of the texture map. We noted the use of magnification and minification filters earlier, and these allow you to address this issue.

In a different direction, the Chromadepth™ 1D texture-mapping process gives excellent 3D effects but does not allow the use of color as a way of encoding and communicating information. It should only be used when the shape alone carries the information that is important in an image, but

it has proved to be particularly useful for geographic and engineering images, as well as molecular models.

Code examples

A 1D color ramp: Sample code to use texture mapping in the Chromadepth™ example is shown below. The declaration set up the color ramp, define the integer texture name, and create the array of texture parameters.

```
float D1, D2;
float texParms[4];
static GLuint texName;
float ramp[256][3];
```

In the `init()` function we find the following function calls that define the texture map, the texture environment and parameters, and then enables the texture generation and application.

```
makeRamp();
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexImage1D(GL_TEXTURE_1D,0,3,256,0,GL_RGB,GL_FLOAT,ramp);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_1D);
```

The `makeRamp()` function is defined to create the global array `ramp[]` that holds the data of the texture map. This process works with the HSV color model in which hues are defined through angles (in degrees) around the circle which has saturation and value each equal to 1.0. The use of the number 240 in the function comes from the fact that in the HSV model, the color red is at 0 degrees and blue is at 240 degrees, with green between at 120 degrees. Thus an interpolation of fully-saturated colors between red and blue will use the angles between 0 and 240 degrees. The RGB values are calculated by a function `hsv2rgb(...)` that is a straightforward implementation of standard textbook color-model conversion processes. The Foley et al. textbook in the references is an excellent resource on color models.

```
void makeRamp(void)
{
   int i;
   float h, s, v, r, g, b;

// color ramp for 1D texture:
// starts at 0, ends at 240, 256 steps
   for (i=0; i<256; i++) {
      h = (float)i*240.0/255.0;
      s = 1.0; v = 1.0;
      hsv2rgb( h, s, v, &r, &g, &b );
      ramp[i][0] = r; ramp[i][1] = g; ramp[i][2] = b;
   }
}
```

Finally, in the `display()` function we find the code below, where `ep` is the eye point parameter used in the `gluLookAt(...)` function. This controls the generation of texture coordinates, and binds the texture to the integer name `texName`. Note that the values in the `texParms[]` array, which define where the 1D texture is applied, are defined based on the eye point, so that the image

will be shaded red (in front) to blue (in back) in the space whose distance from the eye is between D1 and D2.

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
D1 = ep + 1.0; D2 = ep + 10.0;
texParms[0] = texParms[1] = 0.0;
texParms[2] = -1.0/(D2-D1);
texParms[3] = -D1/(D2-D1);
glTexGenfv( GL_S, GL_EYE_PLANE, texParms);
glBindTexture(GL_TEXTURE_1D, texName);
```

An image on a surface: Sample code to use texture mapping in the second example is shown in several pieces below. To begin, in the data declarations we find the declarations that establish the internal texture map (`texImage`) and the set of texture names that can be used for textures (`texName`).

```
#define TEX_WIDTH 512
#define TEX_HEIGHT 512
static GLubyte texImage[TEX_WIDTH][TEX_HEIGHT][3];
static GLuint texName[1]; // parameter is no. of textures used
```

In the init function we find the `glEnable` function that allows the use of 2D textures.

```
glEnable(GL_TEXTURE_2D);  // allow 2D texture maps
```

You will need to create the texture map, either through programming or by reading the texture from a file. In this example, the texture is read from a file named `myTexture.rgb` that was simply captured and translated into a raw RGB file, and the function that reads the texture file and creates the internal texture map, called from the init function, is

```
void setTexture(void)
{
   FILE * fd;
   GLubyte ch;
   int i,j,k;

   fd = fopen("myTexture.rgb", "r");
   for (i=0; i<TEX_WIDTH; i++)     // for each row
   {
      for (j=0; j<TEX_HEIGHT; j++) // for each column
      {
         for (k=0; k<3; k++) // read RGB components of the pixel
         {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
         }
      }
   }
   fclose(fd);
}
```

Finally, in the function that actually draws the cube, called from the `display()` function, we first find code that links the texture map we read in with the texture number and defines the various parameters of the texture that will be needed to create a correct display. We then find code that draws the face of the cube, and see the use of texture coordinates along with vertex coordinates. The vertex coordinates are defined in an array `vertices[]` that need not concern us here.

```
glGenTextures(1, texName);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, TEX_WIDTH, TEX_HEIGHT,
             0, GL_RGB, GL_UNSIGNED_BYTE, texImage);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glBegin(GL_QUADS);
   glNormal3fv(normals[1]);
   glTexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
   glTexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
   glTexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
   glTexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
glEnd();
glDeleteTextures(1,texName);
```

An environment map: The third example also uses a 2D texture map, modified in Photoshop to have a fish-eye distortion to mimic the behavior of a very wide-angle lens. The primary key to setting up an environment map is in the texture parameter function, where we also include two uses of the `glHint(...)` function to show that you can define really nice perspective calculations and point smoothing—with a computation cost, of course. But the images in Figure 11.5 suggest that it might be worth the cost sometimes.

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
...
// the two lines below generate an environment map in both the
// S and T texture coordinates
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

Using multitextures

When we introduced the way OpenGL defines multitextures, we hinted at the kinds of changes you would need to make to use multitextures. Here we will give that code in some detail for a case using two textures so you may see what it would look like.

The declaration of the `textures[]` array would be:

```
int textures[2];
```

In an initialization function we might find the following definitions of the texture objects:

```
// load and bind the textures
glGenTextures(2, &textures);

// load the first texture data into a temporary array
file.open("tex0.raw");
file.read(textureData, 256*256*3);
file.close();

// build the first texture
glBindTexture(GL_TEXTURE_2D, texture[0]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
   GL_NEAREST_MIPMAP_LINEAR);
```

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256,256, GL_RGB,
   GL_UNSIGNED_BYTE, textureData);

// load the second texture data into a temporary array
file.open("tex1.raw");
file.read(textureData, 256*256*3);
file.close();

// build the second texture
glBindTexture(GL_TEXTURE_2D, texture[1]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
   GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256,256, GL_RGB,
   GL_UNSIGNED_BYTE, textureData);
```

In `display()`, we might find the following definitions of the texture units:

```
// set the texture to the first one then bind the texture
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[0]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// set the texture to the second one then bind the texture
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[1]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

And in `display()` or another function that actually implements the geometry of your model, you might find the following code that associates both sets of texture coordinates to each vertex:

```
glBegin(GL_TRIANGLE_STRIP);
  glMultiTexCood2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
  glMultiTexCood2fARB(GL_TEXTURE1_ARB, 0.0, 0.0);
  glVertex3f(-5.0,-5.0,0.0);
  glMultiTexCood2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
  glMultiTexCood2fARB(GL_TEXTURE1_ARB, 0.0, 1.0);
  glVertex3f(-5.0,5.0,0.0);
  glMultiTexCood2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
  glMultiTexCood2fARB(GL_TEXTURE1_ARB, 1.0, 0.0);
  glVertex3f(5.0,-5.0,0.0);
  glMultiTexCood2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
  glMultiTexCood2fARB(GL_TEXTURE1_ARB, 1.0, 1.0);
  glVertex3f(5.0,5.0,0.0);
glEnd();
```

Questions

1. Think about how you would achieve certain effects if you had multitexturing. Begin with some simple effects, such as distressed wood or bullet holes in metal, and think up some new effects and decide how you would create them. If you have multitexturing capability with your graphics API, then implement your ideas.

Exercises

2. Make a number of texture maps from easily-available sources. Use as many of the following techniques as possible: (a) digital photographs, (b) scanned photographs, (c) screen captures, and (d) capturing the contents of the frame buffer from an OpenGL program. In each case, be sure the texture map is saved in a format that can be read into a texture array for your programs.

3. Consider a triangle in 2D space with coordinates V0=(,), V1=(,), and V2=(,) counterclockwise, and take the point P = (,) = α*V0 + β*V1 + δ*V2 for some different values of the coefficients α, β, and δ with $\alpha + \beta + \delta = 1$. If the texture coordinates for the triangle's vertices are T0=(,), T1=(,), and T2=(,) for V0, V1, and V2 respectively, calculate the texture coordinates for the point P. For the choices of coefficients for which these coordinates are not integers, discuss how you would calculate the actual color for point P in terms of colors for nearby points in texture space.

4. Create a different synthetic texture than the one shown in Figure 11.2. For example, you could choose a set of random points in 2D integer space and a sequence of random colors, and for each point in texture space, you could give that point a color that depends on the point to which it is nearest. Or you could use some sort of 2D function pseudocoloring as was discussed in the chapter on science applications. Then use this synthetic texture as we did the checkerboard texture and see what visual results you get.

Experiments

5. Using the 1D texture map concept introduced for the ChromaDepth™ process, define 1D texture maps that could be used to show the elevation on a height field, as introduced in the science examples chapter; extend this to include contour mapping.

6. Use a generally-available program such as POVRay and experiment with the texture options it provides. The goal is to get a sense of how these textures look, especially textures built on the noise function.

7. Pick a texture map that allows you to see the fine details of the texture, such as a checkerboard texture with relatively small squares, and map it onto the GLU quadric objects. Look for points where the texture map on the surface behaves unusually, and see if you can identify those points with any particular geometry on the surface.

8. Take some of the example textures you created in the first exercise above and apply them to the other faces of the cube in Figure 11.13 whose code was given above.

9. Use an artificial texture such as a black-and-white checkerboard pattern to experiment with the use of texture mapping on intensity, luminance, or blending. The results should show the checkerboard pattern, but it should be visible only in the effects on the polygon color instead of by the checkerboard itself being visible.

10. Take an image and create a second image by doing a fisheye converstion of the first using a tool such as Photoshop. Apply both images to some smooth geometry using environment map techniques and discuss the results.

Chapter 12: Dynamics and Animation

Prerequisites

A knowledge of modeling, viewing, lighting, and of the roles of parameters in setting up modeling and viewing.

Introduction

This is an unusual chapter, because it includes few figures that really illustrate its topic. The topic is motion, and we cannot readily capture motion in a written document. It would be possible to include movie files in a purely electronic document, of course, but these notes are intended for print. Perhaps future versions of this will include inline movies, or at least pre-compiled executables with animation, but for now you must work with the examples and code segments that we provide and see the execution on your own systems.

Computer animation is a very large topic, and there are many books and courses on the subject. We cannot hope to cover the topic in any depth in a first course in computer graphics, and indeed the toolkits needed for a great deal of computer animation are major objects of study and skill development in themselves. Instead we will focus on relatively simple animations that illustrate something about the kind of models and images we have been creating in this course, with an emphasis on scientific areas.

Animation is thought of as presenting a sequence of frames, or individual images, rapidly enough to achieve the sense that the objects in the frames are moving smoothly. There are two kinds of animation — real-time animation, or animation in which each frame is presented by the program while it is running, and frame-at-a-time animation, or animation that is assembled by rendering the individual frames and assembling them into a viewable format (possibly through film or video in a separate production process). This chapter focuses more on frame-at-a-time animation with a goal of achieving real-time animation. The two share the problems of defining how models, lighting, and viewing change over time, but frame-at-a-time animation tends to focus on much more detailed modeling and much more sophisticated rendering while real-time animation tends to focus on simpler time-varying information in order to get refresh rates that are high enough to convey the variation that is desired. While real-time animation may not be as realistic as frame-at-a-time animation because simpler modeling and rendering are used or images may be provided at a slower rate, it can be very effective in conveying an idea and can be especially effective if the user can interact with the animation program as it is running.

As with everything else in this course, the real question is visual communication, and there are some special vocabularies and techniques in using animation for communication. This module does not try to cover this in any depth, but we suggest that you spend some time looking at successful animations and trying to discover for yourself what makes them succeed. To start, we suggest that you focus on clarity and simplicity, and work hard to create a focus on the particular ideas you want to communicate.

Definitions

Animation is the process of creating a sequence of images and presenting them so that the viewer's eye will see them as occurring in a smooth motion sequence. The motion sequence can illustrate the relationship between things, can show processes for assembling objects, can allow you to design a sequence of ways to present information, or can allow a user to see a scene from a variety of viewpoints that you can design.

There are many ways to design an animation sequence, but a good place to start is to model your scene using parameters to control features of the model. When you use parameters — variables that you can manipulate in your program — to control positions of objects, positions or properties of lights, shapes or relationships or objects, colors, texture coordinates, or other key points in your model, you can change the values of the parameters with time to change the view you present your audience as the program runs. This allows you to emphasize special features of any of these facets of your model and to communicate those features to your audience.

In defining your modeling in terms of parameters, we need to recall that there are really only three parts to a scene as described by a scene graph. One is the geometry of the scene, where we could use parameters to define the geometry of the scene itself; one example of this could be a function surface, where the function includes a parameter that varies with time, such as $z = \cos(x^2 + y^2 + t)$. Another is the transformations in the scene, where we could use parameters to define the rotation, translation, or scaling of objects in the scene; one example of this could be moving an object in space by a translation with horizontal component t and vertical component $(2 - t)^2$, which would give the object a parabolic path. A third is the appearance of an object in a scene, where a surface might have an alpha color component of (1-t) to change it from opaque at time 0 to transparent at time 1, allowing the user to see through the surface to whatever lies below it. These are straightforward kinds of applications of parametric modeling and should pose no problem in setting up a model.

One way to design an animation sequence is by explicitly controlling the change in the parameters above through your code. This is called *procedural* animation, and it works well for simple animations where you may have only a few parameters that define the sequence (although what defines "a few" may depend on your system and your goals in the sequence). Most of the animation we have discussed or presented in the science applications is procedural, where we compute the positions or behaviors of objects over time from scientific principles and display them as they vary. This direct computation of the properties of each frame of the animation is what distinguishes procedural animation.

Another way to design an animation sequence is by creating key frames, or particular images that you want to appear at particular times in the animation display. Animation done in this way is called *keyframe* animation. Again, each key frame can be defined in terms of a set of parameters that control the display, but instead of controlling the parameters programmatically, the parameters are interpolated between the values at the key frames.

Yet a third kind of animation, though it does not seem to be thought of in those terms, is an *interpolation* animation. Here you define two models and you interpolate the geometry of the first model into the geometry of the second model. One example of this is morphing, where you start with one object (face, animal, automobile, ...) and you end with another, to emphasize the change from one thing to another. This involves a sequence of images, so it is animation, but it is a rather specialized kind of processing and so will not be discussed further here.

Probably the simplest approach to animation is to define your entire scene in terms of a single parameter, and to update that parameter each time you generate a new frame. You could think of the parameter as time and think of your animation in terms of a model changing with time. This is probably a natural approach when you are working with scientific problems, where time plays an active role in much of the modeling — think of how much of science deals with change per unit time. If you know how long it will take to generate your scene, you can even change a time parameter by that amount for each frame so that the viewer will see the frames at a rate that approximates the real-time behavior of the system you are modeling.

Another meaning for the parameter could be frame number, the sequence number of the particular image you are computing in the set of frames that will make up the animation sequence. If you are dealing with animation that you will record and playback at a known rate (usually 24 or 30 frames per second) then you can translate the frame number into a time parameter, but the difference in names for the parameter reflects a difference in thinking, because you will not be concerned about how long it takes to generate a frame, simply where the frame is in the sequence you are building.

A key concept in generating animations in real time is the *frame rate* — the rate at which you are able to generate new images in the animation sequence. As we noted above, the frame rate will probably be lower for highly-detailed generated frames than it would be for similar frames that were pre-computed and saved in digital or analog video, but there's one other difference: frame rates may not be constant for real-time generated images. This points out the challenge of doing your own animations and the need to be sure your animations carry the communication you need. However, the frame rate can be controlled exactly, no matter how complex the images in the individual frames, if you create your own video "hardcopy" of your animation. See the hardcopy chapter for more details on this.

Keyframe animation

When you do a keyframe animation, you specify certain frames as key frames that the animation must produce and you calculate the rest of the frames so that they move smoothly from one key frame to another. The key frames are specified by frame numbers, so these are the parameter you use, as described above.

In cartoon-type animation, it is common for the key frames to be fully-developed drawings, and for the rest of the frames to be generated by a process called "tweening" — generating the frames between the keys. In that case, there are artists who generate the in-between frames by re-drawing the elements of the key frames as they would appear in the motion between key frames. However, we are creating images by programming, so we must start with models instead of drawings. Our key frames will have whatever parameters are needed to defing the images, then, and we will create our in-between frames by interpolating those parameters.

Our animation may be thought of as a collection of animation sequences (in the movies, these are thought of as *shots*), each of which uses the same basic parts, or components, of objects, transformations, lights, etc. For any sequence, then, we will have the same components and the same set of parameters for these components, and the parameters will vary as we go through the sequence. For the purposes of the animation, the components themselves are not important; we need to focus on the set of parameters and on the ways these parameters are changed. With a keyframe animation, the parameters are set when the key frames are defined, and are interpolated in some fashion in moving between the frames.

As an example of this, consider a model that is defined in a way that allows us to identify the parameters that control its behavior. Let us define ...

Figure 13.1: the object we will animate

In order to discuss the ways we can change the parameters to achieve a smooth motion from one key frame to another, we need to introduce some notation. If we consider the set of parameters as a single vector $\boldsymbol{P} = \langle a, b, c, \ldots, n \rangle$, then we can consider the set of parameters at any given frame M as $\boldsymbol{P}_M = \langle a_M, b_M, c_M, \ldots, n_M \rangle$. In doing a segment of a keyframe animation sequence starting with frame K and going to frame L, then, we must interpolate

$\boldsymbol{P}_K = \langle a_K, b_K, c_K, \ldots, n_K \rangle$ and $\boldsymbol{P}_L = \langle a_L, b_L, c_L, \ldots, n_L \rangle$,

the values of the parameter vectors at these two frames. In the example above, we see that the parameters ...

A first approach to this interpolation would be to use a linear interpolation of the parameter values. So if we the number of frames between these key frames, including these frames, is $C = L - K$, we would have $p_i = (ip_K + (C - i)p_L)/C$ for each parameter p and each integer i between L and K. If we let $t = i/C$, we could re-phrase this as $p_i = (tp_K + (1-t)p_L)$, a more familiar way to express pure linear interpolation. This is a straightforward calculation and would produce smoothly-changing parameter values which should translate into smooth motion between the key frames.

Key frame motion is more complex than this simple first approach would recognize, however. In fact, we not only want smooth motion between two key frames, but we want the motion from before a key frame to blend smoothly with the motion after that key frame. The linear interpolation discussed above will not accomplish that, however; instead, we need to use a more general interpolation approach, called easing into and out of the motion. One approach is to start the motion from the starting point more slowly, move more quickly in the middle, and slow down the ending part of the interpolation so that we stop changing the parameter (and hence stop any motion in the image) just as we reach the final frame of the sequence. In Figure 13.2, we see a comparison of simple linear interpolation on the left with a gradual startup/slowdown interpolation on the right. The right-hand figure shows a sinusoidal curve that we could readily describe by $t = 0.5(1 - \cos(f/\pi))$, where in both cases we use $t = i/C$ as in the paragraph above, so that we are at frame K when t=0 and frame L when t=1.

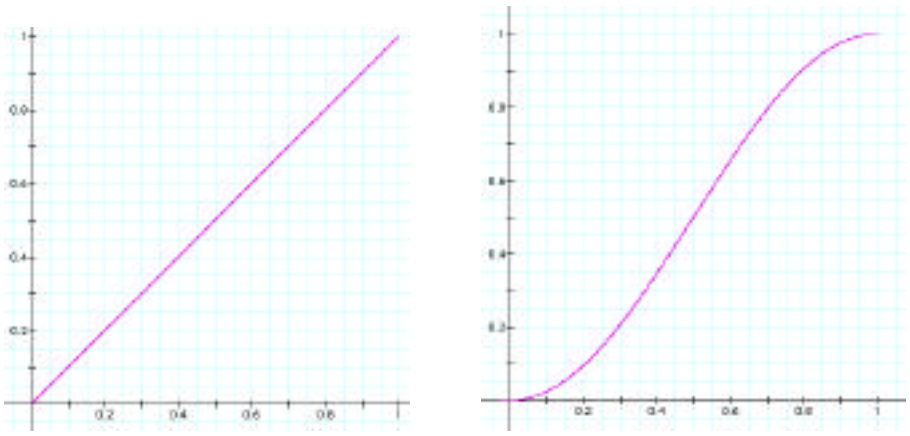


Figure 13.2: two interpolation curves; linear (left) and sinusoidal (right)

In fact, this easing into and easing out of the motion may not be enough, because in order to emphasize the motion, you may want to have the thing you are moving actually back up slightly before it moves forward, and go beyond its final position before it comes to rest. You can adapt

the ideas above to do this by having the interpolation curve move slightly negative as it begins and go slightly above 1 just before it ends. This kind of motion subtlety is where animation becomes art, and we cannot offer sound guidelines on when to use it—except that you should use it when it works.

In spite of our adjustment to move through the key frames slowly, we still have the problem that a parameter can provide motion (or another effect) in one direction up to a key frame, and then that motion or effect can go off in an entirely different direction when it leaves the key frame and goes to another one. That is, our motion is not yet smooth as it goes through a key frame. To achieve this, we will need to provide a more sophisticated kind of interpolation.

If we consider the quadratic and cubic interpolations from the mathematical fundamentals in an early chapter, we see that there are interpolations among a number of points that meet some of the points. We need to find such an interpolation that allows our interpolation to meet each keyframe exactly and that moves smoothly among a set of keyframes, and in order to do this we need to be able to interpolate the parameters for the frames in a way that meets exactly the parameters for the keyframes and that moves smoothly between values of the parameters. From the various kinds of interpolations available, we would chose the Catmull-Rom interpolation described in the chapter on spline modeling, which gives us the kind of interpolation shown in the second row of Figure 13.3 and that contrasts with the point-to-point interpolation shown in the first row of that figure.

Figure 13.3: moving in and out of a keyframe (left to right follows time).
Top row: using a two-point interpolation; bottom row: using a multi-point interpolation

Temporal aliasing

In creating an animation, you are creating a sequence of images that represent the state of your model at specific points in time. When these sequences are viewed in order, however, you may find that the results show some surprising effects that you did not intend. Some of these problems may be due to problems with the graphics system; for example, if you have a very small object, it may seem to vary in size over time as more or fewer pixels get chosen for the object. This is a screen aliasing problem and can be addressed by using antialiasing techniques to incluse partly-covered pixels in the image. But other problems are fundamental in the animation process and cannot be readily eliminated; you must recognize the possibility that your images in sequence may be interpreted differently than you intended.

Let's consider an example. Suppose you have an object like that shown in Figure 13.4 (for example, this might be the spokes of a wheel) and rotate the figure with time. If you rotate the figure slowly, the eye will naturally follow each spoke as it changes position because the position of the spoke in the next frame will be the one nearest that spoke in the previous frame. This will happen if you want to show the spokes moving clockwise and you rotate the figure clockwise by an angle less than 22.5°, particularly if the angle is much less than that value. But if you rotate the

figure a bit more quickly, say by 40°, then the position of the spoke in the next frame will be only 5° from the position of the next clockwise spoke in the previous frame (read that phrase over again to be sure what we're saying!) and the spokes will seem to be rotating counterclockwise. You may have seen this happen in films or when a rotating object is lit by a strobe light, but it is important to realize that it is possible and to plan to manage it in your work.

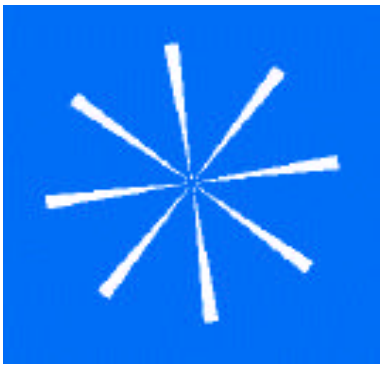


Figure 13.4: an object that might be rotated

Building an animation

The communication you are developing with your animation is very similar to the communication that a director might want to use in a film. Film has developed a rich vocabulary of techniques that will give particular kinds of information, and animations for scientific communication can benefit from thinking about issues in cinematic terms. If you plan to do this kind of work extensively, you should study the techniques of professional animators. Books on animations will show you many, many more things you can do to improve your planning and execution. Initially, you may want to keep your animations simple and hold a fixed eyepoint and fixed lights, allowing only the parts of the model that move with time to move in your frames. However, just as the cinema discovered the value of a moving camera in a moving scene when they invented the traveling shot, the camera boom, and the hand-held walking camera, you may find that you get the best effects by combining a moving viewpoint with moving objects. Experiment with your model and try out different combinations to see what tells your story best.

A word to the wise...

Designing the communication in an animation is quite different from the same task for a single scene, and it takes some extra experience to get it really right. Among the techniques used in professional animation is the storyboard — a layout of the overall animation that says what will happen when as the program executes, and what each of your shots is intended to communicate to the viewer.

Some examples

Moving objects in your model

Since animation involves motion, one approach to animation is to move individual things in your model. We may take a mathematical approach to defining the position of a cube, for example, to move it around in space. This is done in the idle event callback function `animate()` that we show here:

```
void animate(void)
```

```
{
   #define deltaTime 0.05

// define position for the cube by modeling time-based behavior
   aTime += deltaTime; if (aTime > 2.0*PI) aTime -= 2.0*PI;
   cubex = sin(2.0*aTime);
   cubey = cos(3.0*aTime);
   cubez = cos(aTime);
   glutPostRedisplay();
}
```

This function sets the values of three variables that are later used as the parameters for the transformation `glTranslate*(...)` that positions the cube in space. You could similarly use other transformations with parameters to set variable orientation, size, or other properties of your objects as well.

Moving parts of objects in your model

Just as we moved whole objects above, you could move individual parts of a heirarchial model. You could change the relative positions, relative angles, or relative sizes by using variables when you define your model, and then changing the values of those variables in the idle callback. You can even get more sophisticated and change colors or transparency when those help you tell the story you are trying to get across with your images. The code below increments the parameter `t` and then uses that parameter to define a variable that is, in turn, used to set a rotation angle to wiggle the ears of the rabbit head described in the discussion of hierarchical modeling. In that discussion we developed the scene graph for the rabbit's head, as in Figure 13.5. Here we note that there are transformations that place the ears on the head, but we do not yet define them in detail; this is done in the code below.

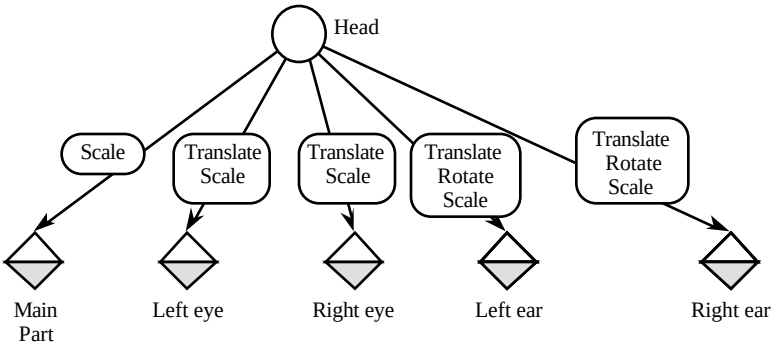


Figure 13.5: the scene graph for the rabbit's head

The code that defines one of the rabbit's ears is as follows. Note that there are a number of transformations involved in this process, but only two of them involve the parameter `wiggle` that moves the ears from frame to frame; the others are fixed transformations that do not change from frame to frame.

```
glPushMatrix();
      //    model the left ear
      glColor3f(1.0, 0.6, 0.6);     //    pink ears
      glRotatef(-10.0*wiggle, 0.0, 0.0, 1.0);
      glTranslatef(-1.0, -1.0, 1.0);
      glRotatef(-45.0, 1.0, 0.0, 0.0);
      glTranslatef( 0.5, 0.0, 0.0); // begin
```

```
            glRotatef(-10.0*wiggle, 0.0, 0.0, 1.0);
            glTranslatef(-0.5, 0.0, 0.0); // end
            glScalef(0.5, 2.0, 0.5);
            myQuad = gluNewQuadric();
            gluSphere(myQuad, 1.0, 10, 10);
      glPopMatrix();
```

The idle callback that manipulates the parameter `wiggle` that controls the angle at which the ears are placed on the head is given by the following:

```
void animate(void)
{
#define twopi 6.28318

   t += 0.1;
   if (t > twopi) t -= twopi;
   wiggle = cos(t);
   glutPostRedisplay();
}
```

Moving the eye point or the view frame in your model

Another kind of animation is provided by providing a controlled motion around a scene to get a sense of the full model and examine particular parts from particular locations. This motion can be fully scripted or it can be under user control, though of course the latter is more difficult. In this example, the eye moves from in front of a cube to behind a cube, always looking at the center of the cube, but a more complex (and interesting) effect would have been achieved if the eye path were defined through an evaluator with specified control points. This question may be revisited when we look at evaluators and splines.

```
void display( void )
{
// Use a variable for the viewpoint, and move it around ...
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt( ep.x, ep.y, ep.z, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
...
}

void animate(void)
{
   GLfloat numsteps = 100.0, direction[3] = {0.0, 0.0, -20.0};

   if (ep.z < -10.0) whichway = -1.0;
   if (ep.z > 10.0)  whichway = 1.0;
   ep.z += whichway*direction[2]/numsteps;
   glutPostRedisplay();
}
```

As you travel, you need to control not only the position of the eye point, but also the entire viewing environment — in simple terms, the entire `glutLookAt(...)` parameter list. So not only the eye point, but also the view reference point and the up vector must be considered in creating an effective moving viewpoint. Further, as you move around you will sometimes find yourself moving nearer to objects or farther from them. This means you will have the opportunity to use level-of-detail techniques to control how those objects are presented to the viewer while you keep

the frame rate as high as possible. There's a lot of work here to do everything right, but you can make a good start much more easily.

Changing features of your model

There are many other special features of your models and displays that you can change with time to create the particular communication you want. Among them, you can change colors, properties of your lights, transparency, clipping planes, fog, texture maps, granularity of your model, and so on. Almost anything that can be defined with a variable instead of a constant can be changed by changing the model.

In the particular example for this technique, we will change the size and transparency of the display of one kind of atom in a molecule, as we show in Figure 13.6. The change in the image is driven by a parameter `t` that is changed in the `idle` callback, and the parameter in turn gives a sinusoidal change in the size and transparency parameters for the image. This will allow us to put a visual emphasis on this kind of atom so that a user could see where that kind of atom fits into the molecule. This is just a small start on the overall kinds of things you could choose to animate to put an emphasis on a part of your model.

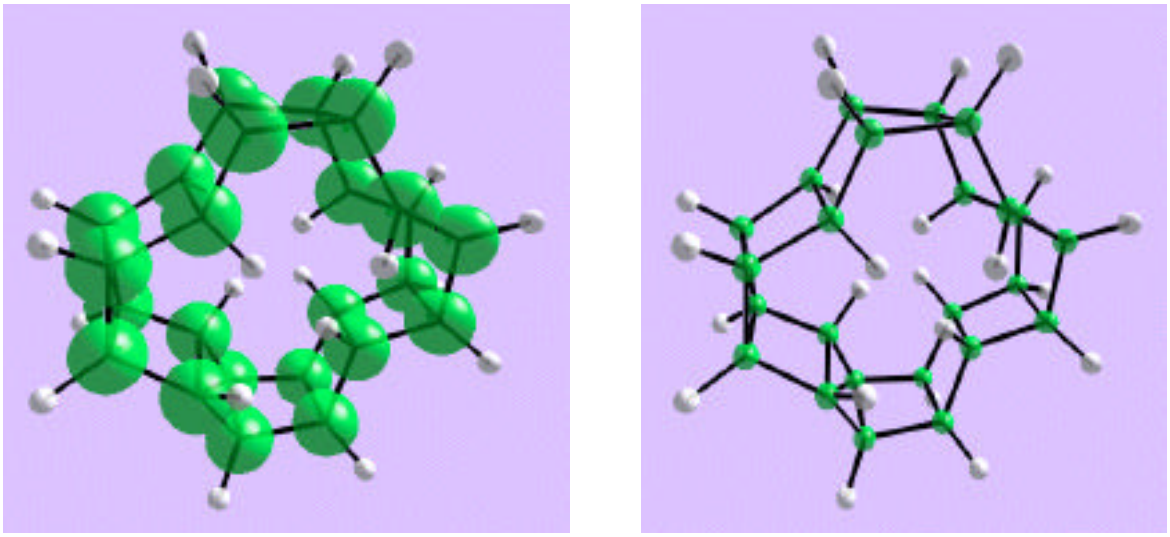


Figure 13.6: molecule with carbon expanded (left) or contracted (right)

Code to carry out this operation is shown below.

```
void molecule(void)
{
...
         j = atoms[i].colindex;  // index of color for atom i
         for (k=0; k<4; k++)
         {  // copy atomColors[j], adjust alpha by alphaMult
            myColor[k] = atomColors[j][k];
         }
         if (j==CARBON) myColor[3] += alphaAdd;
         glMaterialfv(..., myColor );
         glTranslatef(...);
         if (j==CARBON)
            gluSphere(atomSphere,sizeMult*ANGTOAU(atomSizes[j]),GRAIN,GRAIN);
         else
```

```
            gluSphere(atomSphere,ANGTOAU(atomSizes[j]),GRAIN,GRAIN);
         glPopMatrix();
...
 }
...
void animate(void)
{
   t += 0.1; if (t > 2.0*M_PI) t -= 2.0*M_PI;
   sizeMult = (1.0+0.5*sin(t));
   alphaAdd = 0.2*cos(t);
   glutPostRedisplay();
}
```

Some points to consider when doing animations with OpenGL

There are some things about OpenGL that you need to understand more fully when you move your eyepoint than you when you simply create a single image. The viewing transformation is part of the overall modeling transformation, and it needs to be done at the right place if you are going to use parameters to define your view. In the `display()` function in the `viewcube.c` example, you will note that the modeling transformation is set to the identity, the `glutLookAt(...)` function is called, the resulting transformation is saved for future use, and then the rotation processes are called. This keeps the viewing transformation from changing the position of objects in the model and thus keeps the animation looking right.

Finally, be careful when you use texture maps in animations. There is always the possibility of aliasing with texture maps, and when they are animated the aliasing can cause strange-looking behavior in the texture rendering. Some effort in antialiasing textures is particularly important in animating them.

Code examples

As we noted above, and as the code examples show, animation control is primarily managed by changing parameters of your scene in the callback for the `idle` event. We have seen several of these examples to control several aspects of the model, the scene, and the display. You should experiment as widely as you can to see what kind of things you can control and how you can control them in order to become fluent in using animation as a communication tool.

A word to the wise

You should look at videos of computer graphics work to get a fuller understanding of what you can do with this tool. In general, though, you want to avoid high-end entertainment-focused animations and look at informational animations—presentations of scientific or technical work are ideal. But remember that when you look at video animatinos, you are probably looking at *presentation-level* animations, or work that is done to impress others with the concepts being presented. Such work is usually very highly designed and involves a great deal of sophisticated thinking and high-end graphics systems and tools. The work you will do in a beginning computer graphics course is much more likely to be *personal-level* or *peer-level* animation: work that is done to explore an idea yourself or to share with a friend or colleague. Our experience is that even these animations can be very valuable; a number of people in the sciences have asked for copies of student animations that have been very effective at illustrating concepts in the sciences. So don't try to match the quality of the videos you watch; try to find the key communication ideas in the videos and learn from them, and your work will be valuable.

Chapter 13: High-Performance Graphics Techniques

Prerequisites

A solid understanding of the concepts of computer graphics, and a knowledge of the details of the OpenGL graphics API in sufficient depth that you can consider alternatives to standard approaches to creating animated, interactive images.

Definitions

The speed with which we can generate an image is always of interest because we want to be able to get our results quickly, whether we're doing a single scene or an animation. Waiting for an image to be produced can be frustrating and can even get in the way of the image's effectiveness. This is evident in many kinds of graphics applications, but it is probably most evident in computer games, so this is the context that will frame much of the discussion in this chapter. However, performance is always an issue in graphical computations, so the techniques we discuss here are also important to many other graphics applications.

Making effective computer games involves many things, including storytelling, creating characters, maintaining databases of game features, and many general programming techniques to deliver maximum speed of operation to the player. One of the critical performance areas—because it's one of the most compute-intensive bottlenecks in presenting the game to the player—is the graphics that present the game to the user. This is a different question than we've been dealing with in computer graphics to this point. Up to this point in these notes, we have focused on the quality of the images while maintaining as much performance as we can, but in this chapter we reverse this emphasis: we focus on the performance of the programs while maintaining as much quality as we can. This change makes a major difference in the kind of processes and techniques we use.

In a sense, this is not a new issue in computer graphics. For over 20 years, the computer field has included the area of "real-time graphics." This originally focused on areas such as flight simulators, real-time monitoring of safety-critical system processes, and real-time simulations, often using the highest-power computers available at the time. Some of the real-time graphics processes also were used in educational applications that are essentially simulations. But the demands that games place on graphics are both as extreme as these and are focused on personal computers with widely varying configurations, making it important to bring these real-time techniques into a graphics text.

Techniques

Fundamentally, high-performance computer graphics, especially as applied to games, takes advantage of a few simple principles:

- Use hardware acceleration when you can, but don't assume that everyone has it and be ready to work around it when you need to
- Do some work to determine what you don't need to display
    - Look for techniques that will support easy ways to cull objects or pieces of objects from the set of things to be displayed
- Take advantage of capabilities of texture mapping
    - Create objects as texture maps instead of as fully-rendered objects
    - Use multitextures and textures with both low and high resolution
- Use any techniques available to support the fastest display techniques you can
    - Display lists
    - Level of detail
- Avoid unnecessary lighting calculations

    - When you draw any object, only enable lights near the object
    - Use fog
- Collision detection

We will discuss these and show you as many techniques as we can to support them in helping you create high-performance graphics.

There are also some unique problems in some gaming situations that are not found in most other areas of graphics. The main one we will discuss is collision detection, because this is an area that requires some simple computation that we can streamline in ways that are similar to the techniques discussed above.

Hardware avoidance

The computer graphics pipeline includes a number of places where there are opportunities to put hardware into the process to get more performance, and graphics cards have usually been quick to take advantage of these. When you use OpenGL on a system with such a card, the graphics system will probably use the hardware features automatically. Paradoxical as it may seem, however, relying on this approach to speed may not be the best idea for high performance. Parts of your audience might not have the kind of acceleration you are programming for, for example, and even hardware has its cost. But more fundamentally, sometimes using standard techniques and relying on the hardware to get you speed will be slower than looking for alternative techniques that can avoid the processing the card accelerates.

As an example, consider the Z-buffer that is supported on almost every graphics card. When you use the Z-buffer to handle depth testing, you must carry out reading, comparing, and writing operations for each pixel you draw. If you have a fast graphics card, this is higher-speed reading, comparing, and writing, of course, but it is faster to avoid these operations than to optomize them. There are some techniques we will talk about below that allow you to do some modest computation to avoid entire polygons or, even better, to avoid making depth tests altogether.

Designing out visible polygons

As you lay out the overall design of your image, you can ensure that there is only limited visibility of the overall scene from any viewpoint. This is part of the reason why one sees many walls in games. We see large polygons with texture mapping for detail, and only a very few polygons are visible from any point. The sense of visual richness is maintained by moving quickly between places with different and clearly understood environments, so that when the player makes the transition from one place to another, they see a very different world, and even though the world is simple, the constant changing makes the game seem constantly fresh.

Other techniques involve pre-computing what objects will be visible to the player from what positions. As a very simple example, when a player moves out of one space into another, nothing in the space being vacated can be seen, so all the polygons in that space can be ignored. this kind of pre-computed design can involve maintaining lists of visible polygons from each point with each direction the player is facing, a classical tradeoff of space for speed.

Culling polygons

One of the traditional techniques for avoiding drawing is to design only objects that are made up of polyhedra (or can be made from collections of polyhedra) and then to identify those polygons in the polyhedra that are back faces from the eye point. In any polyhedron whose faces are opaque, any polygon that faces away from the eye point is invisible to the viewer, so if we draw these and

use the depth buffer to manage hidden surfaces, we are doing work that cannot result in any visible effects. Thus it is more effective to decide when to avoid drawing these at all.

The decision as to whether a polygon faces toward or away from the eye point is straightforward. Remember that the normal vector for the polygon points in the direction of the front (or outside) face, so that the polygon will be a front face if the normal vector points toward the eye, and will be a back face if the normal vector points away from the eye. Front faces are potentially visible; back faces are never visible. In terms of the diagram in Figure 13.1, with the orientation of the vectors N and E as shown, a front face will have an acute angle between the normal and eye vectors, so N•E will be positive, and a back face will have an obtuse angle, so N•E will be negative. Thus a visibility test is simply the algebraic sign of the term N•E. Choosing not to display any face that does not pass the visibility test is called *backface culling*.

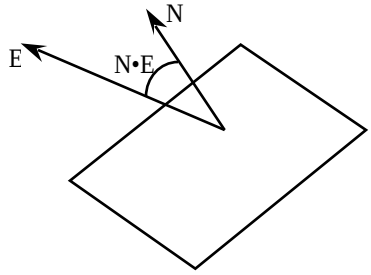


Figure 13.1: the front face test

This kind of culling can readily be done in your graphics program before any calls to the graphics API functions, but many graphics APIs support backface culling directly. In OpenGL, culling is supported by enabling an operational procedure, `GL_CULL_FACE`. Deciding what makes up a front face is done with the function

```
void glFrontFace(Glenum mode)
```

where `mode` takes on the values `GL_CCW` or `GL_CW` (counterclockwise or clockwise), depending on the orientation of the vertices of a front face as seen from the eye point. You can then choose which kind of face to cull with the function

```
void glCullFace(Glenum mode)
```

in this case, `mode` takes on the values `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. If culling is enabled, polygons are not drawn if they are the kind of face selected in `glCullFace`, where the concept of a front face is defined in `glFrontFace.`

Another kind of culling can take place on the viewing volume. Here you can compare each vertex of your polyhedron or polygon with the bounding planes on your view volume; if all of the vertices lie outside of the viewing volume based on comparisons with the same bounding plane, then the polyhedron or polygon cannot be seen in the defined view and need not be drawn. This calculation should be done after the viewing transformation so the boundaries of the view volume are easy to use, but before the polygons are actually rendered. Recalling that the viewing volume is a rectangular pyramid with apex at the origin and expanding in the negative Z-direction, the actual comparison calculations are given by the following:

`y > T*Z/ZNEAR` or `y < B*Z/ZNEAR`
`x > R*Z/ZNEAR` or `x < L*Z/ZNEAR`
`z > ZNEAR` or `z < ZFAR`

where `T`, `B`, `R`, and `L` are the top, bottom, right, and left coordinates of the near plane `Z = ZNEAR` as indicated by the layout in the diagram in Figure 13.2 below.

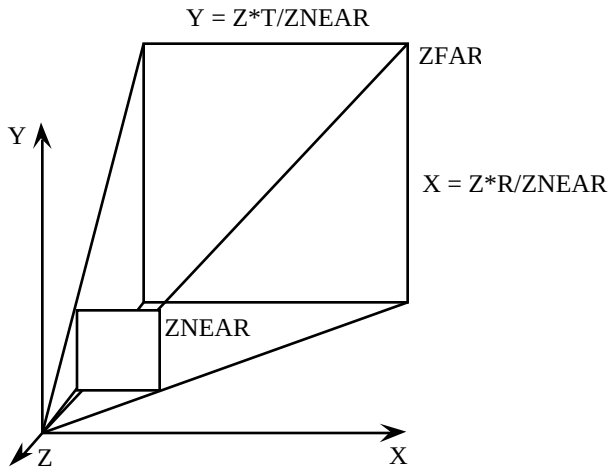


Figure 13.2: the comparisons for the bounding volume computation

Avoiding depth comparisons

One of the classic computer graphics techniques is to order your objects by depth and draw them from back to front, mimicing the way light would progress from objects to your eye. This is called the *painter's algorithm*, and it was most popular when the Z-buffer was beyond the scope of most graphics programming. This technique can be relatively simple if your model is static, had no interlocking polygons, and was intended to be seen from a single viewpoint, because these make it easy to figure out what "back" and "front" mean and which of any two polygons is in front of the other. This is not the usual design philosophy for interactive graphics, however, and particularly for games, because moving geometry and moving eye points are constantly changing which things are in front of what others. So if we were to use this approach, we would find ourselves having to calculate distances from a moving eye point in varying directions, which would be very costly to do.

It may be possible to define your scene in ways that can ensure that you will only view it from points where the depth is known, or you may need to define more complex kinds of computation to give you that capability. A relatively common approach to this problem is given by binary space partitioning, as described below.

Front-to-back drawing

Sometimes a good idea is also a good idea when it is thought of backwards. As an alternative to the painter's algorithm approach, sometimes you can arrange to draw objects only from the front to the back. This still requires a test, but you need test only whether a pixel has been written before you write it for a new polygon. When you are working with polygons that have expensive calculations per pixel, such as complex texture maps, you want to avoid calculating a pixel only to find it overwritten later, so by drawing from the front to back you can calculate only those pixels you will actually draw. You can use BSP tree techniques as discussed below to select the nearest objects, rather than the farthest, to draw first, or you can use pre-designed scenes or other approaches to know what objects are nearest.

Binary space partitioning

There are other approach to avoiding depth comparisons. It is possible to use techniques such as binary space partitioning to determine what is visible, or to determine the order of the objects as seen from the eyepoint. Here we design the scene in a way that can be subdivided into convex

sub-regions by planes through the scene space and we can easily compute which of the sub-regions is nearer and which is farther. This subdivision can be recursive: find a plane that does not intersect any of the objects in the scene and for which half the objects are in one half-space relative to the plane and the other half are in the other half-space, and regard each of these half-spaces as a separate scene to subdivide each recursively. The planes are usually kept as simple as possible by techniques such as choosing the planes to be parallel to the coordinate planes in your space, but if your modeling will not permit this, you can use any plane at all. This technique will fail, however, if you cannot place a plane between two objects, and in this case more complex modeling may be needed. This kind of subdivision is illustrated in Figure 13.3 for the simpler 2D case that is easier to see.

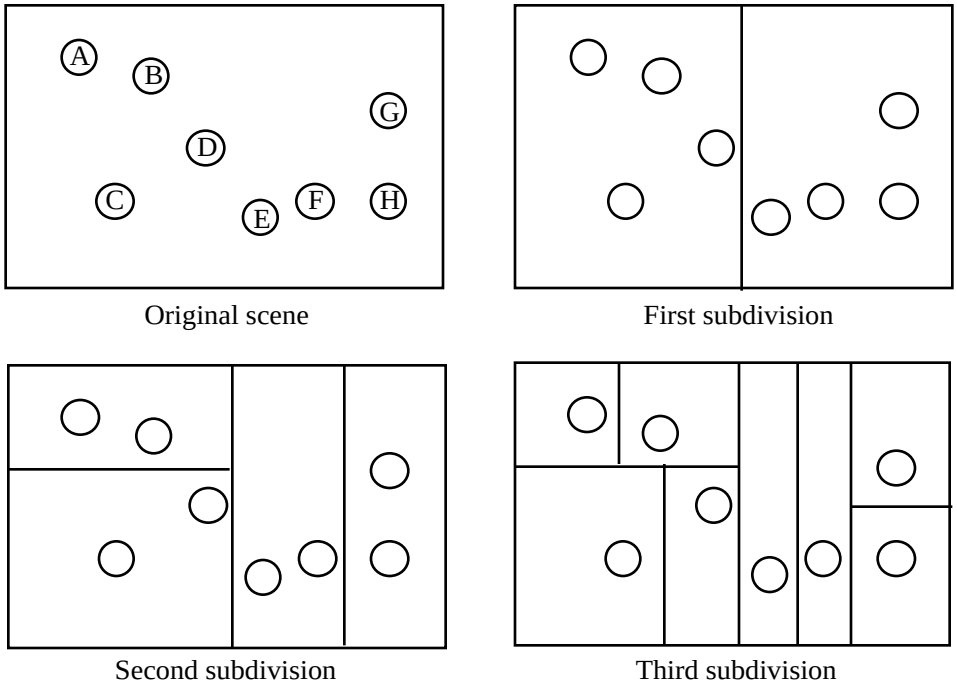


Figure 13.3: a collection of objects in a subdivided space

This partitioning allows us to view the space of the image in terms of a binary space partitioning tree (or *BSP tree*) that has the division planes as the interior nodes and the actual drawn objects as its leaves. With each interior note you can store the equation of the plane that divides the space, and with each branch of the tree you can store a sign that says whether that side is positive or negative when its coordinates are put into the plane equation. This tree is shown in Figure 13.4,

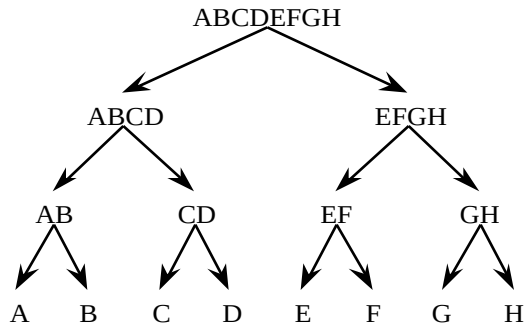


Figure 13.4: a binary space partitioning tree

with each interior node indicated by the letters of the objects at that point in the space. These support the computation of which side is nearer the eye, as noted below. With any eye point, you can determine which parts of the space are in front of which other parts by making one test for each interior node, and re-adjusting the tree so that (for example) the farther part is on the left-hand branch and the nearer part is on the right-hand branch. This convention is used for the tree in the figure with the eye point being to the lower right and outside the space. The actual drawing then can be done by traversing the tree left-to-right and drawing the objects as you come to them.

The actual test for which part is nearer can be done by considering the relation of the eye point to the plane that divides the space. If you put the eye coordinates into the plane equation, you will get either a positive or negative value, and objects on the side of the plane nearer the eye will have the same relation to the plane as the eye. Further, as your eye moves, you will only need to recompute the orientation of the BSP tree when your eye point crosses one of the partitioning planes, and you may be able to conclude that some of the orientations do not need to be recomputed at all.

If you have any moving objects in your scene, you must determine their relation to the other objects and account for them in relation to the BSP tree. It is common to have moving objects only show up in front of other things, and if this is the case then you can draw the scene with the BSP tree and simply draw the moving object last. However, if the moving object is placed among the other drawn objects, you can add it into the BSP tree in particular spaces as it moves, with much the same computation of its location as you did to determine the eye location, and with the object moved from one region to another when it crosses one of the dividing planes. Details of this operation are left to the reader at this time.

Clever use of textures

We have already seen that textures can make simple scenes seem complex and can give an audience a sense of seeing realistic objects. When we take advantage of some of the capabilities of texture mapping we can also deal with graphic operations in precisely the sense that we started this chapter with: reducing the accuracy in hard-to-see ways while increasing the efficiency of the graphics.

One technique is called *billboarding*, and involves creating texture-mapped versions of complex objects that will only be seen at a distance. By taking a snapshot — either a photograph or a once-computed image — and using the alpha channel in the texture map to make all the region outside the object we want to present blend to invisible, we can put the texture onto a single rectangle that is oriented towards the eye point and get the effect of a tree, or a building, or a vehicle, on each rectangle. If we repeat this process many times we can build forests, cities, or parking lots without doing any of the complex computation needed to actually compute the complex object. Orienting each billboard to eye point involves computing the positions of the billboard and the eye (which can be readily done from the scene graph by looking for translations that affect both) and computing the cylindrical or spherical coordinates of the eye point if the billboard is regarded as the origin. The latitude and longitude of the eye point from the billboard will tell you how to rotate the billboard so it faces toward the eye. Note that there are two ways to view a billboard; if it represents an object with a fixed base (tree, building, ...) then you only want to rotate it around its fixed axis; if it represents an object with no fixed point (snowflake) then you probably want to rotate it around two axes so it faces the eye directly.

Another technique is to use techniques at several levels of resolution. OpenGL provides a capacity to do *mipmaps*, texture maps at many resolutions. If you start with the highest-resolution (and hence largest) texture map, you can automatically create texture maps with lower resolution. Recall that each dimension of any texture map must be a power of two, so you can create maps with dimensions half the original, one fourth the original, and so on, yielding a sequence of texture

maps that you can use to achieve your textures without the aliasing you would get if you used the larger texture.

Yet another approach is to layer textures to achieve your desired effects. This capability, called *multitexturing*, is an extension of OpenGL that is found in a number of systems. It allows you to apply multiple textures to a polygon in any order you want, so you can create a brick wall as a color texture map, for example, and then apply a luminance texture map to make certain parts brighter, simulating the effect of light through a window or the brightness of a torch without doing any lighting computations whatsoever. This is discussed in more detail in the later chapter on texture mapping.

These last two techniques are fairly advanced and the interested student is referred to the manuals for more details.

System speedups

One kind of speedup available from the OpenGL system is the display list. As we noted in Chapter 3, you can assemble a rich collection of graphics operations into a display list that executes much more quickly than the original operations. This is because the computations are done at the time the display list is created, and only the final results are sent to the final output stage of the display. If you pre-organize chunks of your image into display lists, you can execute the lists and gain time. Because you cannot change the geometry once you have entered it into the display list, however, you cannot include things like polygon culling or changed display order in such a list.

Another speedup is provided by the "geometry compression" of triangle strips, triangle fans, and quad strips. If you can ensure that you can draw your geometry using these compression techniques, even after you have done the culling and thresholding and have worked out the sequence you want to use for your polygons, these provide significant performance increases.

Level of Detail

Level of detail (usually just called LOD) is a set of techniques for changing the display depending on the view the user needs in a scene. It can involve creating multiple versions of a graphical element and displaying a particular one of them based on the distance the element is from the viewer. It can also involve choosing not to display an object if it is so far from the user that it is too small to display effectively, or displaying a blurred or hazy version of an object if it is not near the eye. LOD techniques allow you to create very detailed models that will be seen when the element is near the viewer, but more simple models that will be seen when the element is far from the viewer. This saves rendering time and allows you to control the way things will be seen, or even whether the element will be seen at all.

Level of detail is not supported directly by OpenGL, so there are few definitions to be given for it. However, it is becoming an important issue in graphics systems because more and more complex models and environments are being created and it is more and more important to display them in real time. Even with faster and faster computer systems, these two goals are at odds and techniques must be found to display scenes as efficiently as possible.

The key concept here seems to be that the image of the object you're dealing with should have the same appearance at any distance. This would mean that the farther something is, the fewer details you need to provide or the coarser the approximation you can use. Certainly one key consideration is that one would not want to display any graphical element that is smaller than one pixel, or perhaps smaller than a few pixels. Making the decision on what to suppress at large distance, or

what to enhance at close distance, is probably still a heuristic process, but there is research work on coarsening meshes automatically that could eventually make this better.

LOD is a bit more difficult to illustrate than fog, because it requires us to provide multiple models of the elements we are displaying. The standard technique for this is to identify the point in your graphical element `(ObjX, ObjY, ObjZ)` that you want to use to determine the element's distance from the eye. OpenGL will let you determine the distance of any object from the eye, and you can determin the distance through code similar to that below in the function that displayed the element:

```
glRasterPos3f( ObjX, ObjY, ObjZ );
glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &dist );
if (farDist(dist)) { ... // farther element definition
}
else { ...               // nearer element definition
}
```

This allows you to display one version of the element if it is far from your viewpoint (determined by the a function `float farDist(float)` that you can define), and other versions as desired as the element moves nearer to your viewpoint. You may have more than two versions of your element, and you may use the distance that

```
glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &dist)
```

returns in any way you wish to modify your modeling statements for the element.

To illustrate the general LOD concept, let's display a GLU sphere with different resolutions at different distances. Recall from the early modeling discussion that the GLU sphere is defined by the function

```
void gluSphere (GLUquadricObj *qobj, GLdouble radius,
                GLint slices, GLint stacks);
```

as a sphere centered at the origin with the radius specified. The two integers *slices* and *stacks* determine the granularity of the object; small values of *slices* and *stacks* will create a coarse sphere and large values will create a smoother sphere, but small values create a sphere with fewer polygons that's faster to render. The LOD approach to a problem such as this is to define the distances at which you want to resolution to change, and to determine the number of slices and stacks that you want to display at each of these distances. Ideally you will analyze the number of pixels you want to see in each polygon in the sphere and will choose the number of slices and stacks that provides that number.

Our modeling approach is to create a function `mySphere` whose parameters are the center and radius of the desired sphere. In the function the depth of the sphere is determined by identifying the position of the center of the sphere and asking how far this position is from the eye, and using simple logic to define the values of *slices* and *stacks* that are passed to the `gluSphere` function in order to select a relatively constant granularity for these values. The essential code is given below, and some levels of the sphere are shown in Figure 13.5.

```
     myQuad=gluNewQuadric();
     glRasterPos3fv( origin );
//   howFar = distance from eye to center of sphere
     glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &howFar );
     resolution = (GLint) (200.0/howFar);
     slices = stacks = resolution;
     gluSphere( myQuad , radius , slices , stacks  );
```

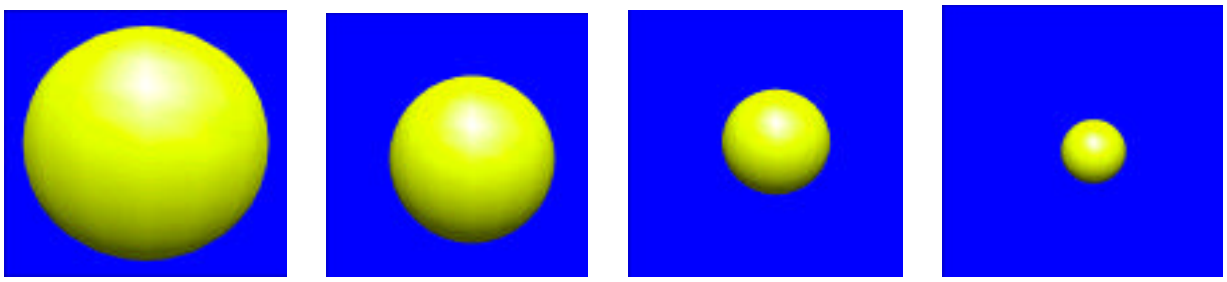


Figure 13.5: levels of detail in the sphere, from high detail level at left to lower at right

As LOD techniques are used in animated or dynamic scenes, you must avoid sudden appearance or disappearance of objects (as they are clipped or unclipped by a distant plane, for example) as well as sudden jumps in objects' appearance. These artifacts cause a break in the action that destroys the believability of the action. It can be useful to create a fog zone deep in a scene and have things appear through the fog instead of simply jumping into place. Fog is discussed below.

Reducing lighting computation

While we may include eight (or more) lights in a scene, each light we add takes a toll on the time it takes to render the scene. Recalling the lighting computations, you will recall that we calculate the ambient, diffuse, and specular lighting for each light and add them together to compute the light for any polygon or vertex. However, if you are using positional lights with attenuation, the amount of light a particular light adds to a vertex is pretty small when that vertex is not near the light. You may choose to simplify the light computation by disabling lights when they are not near the polygon you are working on. Again, the principle is to spend a little time on computation when it can offer the possibility of saving more time on the graphics calculation.

Fog

Fog is a technique which offers some possibility of using simpler models in a scene while hiding some of the details by reducing the visibility of the models. The tradeoff may or may not be worth doing, because the simpler models may not save as much time as it takes to calculate the effect of the fog. We include it here more because of its conceptual similarity to level-of-detail questions than for pure efficiency reasons.

When you use fog, the color of the display is modified by blending it with the fog color as the display is finally rendered from the OpenGL color buffer. Details of the blending are controlled by the contents of the depth buffer. You may specify the distance at which this blending starts, the distance at which no more blending occurs and the color is always the fog color, and the way the fog color is increased through the region between these two distances. Thus elements closer than the near distance are seen with no change, elements between the two distances are seen with a color that fades towards the fog color as the distance increases, and elements farther than the far distance are only seen with the full effect of the fog as determined by the fog density. This provides a method of depth cueing that can be very useful in some circumstances.

There are a small number of fundamental concepts needed to manage fog in OpenGL. They are all supplied through the `glFog*(`*`param`*`, value)` functions as follows, similarly to other system parameter settings, with all the capitalized terms being the specific values used for *`param`*. In this discussion we assume that color is specified in terms of RGB or RGBA; indexed color is noted briefly below.

start and end:

fog is applied between the starting value `GL_FOG_START` and the ending value `GL_FOG_END`, with no fog applied before the starting value and no changes made in the fog after the end value. Note that these values are applied with the usual convention that the center of view is at the origin and the viewpoint is at a negative distance from the origin. The usual convention is to have fog start at 0 and end at 1.

mode:

OpenGL provides three built-in fog modes: linear, exponential, or exponential-squared. These affect the blending of element and fog color by computing the fog factor `ff` as follows:

- GL_LINEAR: `ff = density*z'` for `z'= (end-z)/(end-start)` and any `z` between `start` and `end`.
- GL_EXP: `ff = exp(-density*z')` for `z'` as above
- GL_EXP2: `ff = exp(-density*z')`^{2} for `z'` as above

The fog factor is then clamped to the range [0,1] after it is computed. For all three modes, once the fog factor `ff` is computed, the final displayed color `Cd` is interpolated by the factor of ff between the element color `Ce` and the fog color `Cf` by

`Cd=ff*Ce+(1-ff)*Cf`.

density:

density may be thought of as determining the maximum attenuation of the color of a graphical element by the fog, though the way that maximum is reached will depend on which fog mode is in place. The larger the density, the more quickly things will fade out in the fog and thus the more opaque the fog will seem. Density must be between 0 and 1.

color:

while we may think of fog as gray, this is not necessary — fog may take on any color at all. This color may be defined as a four-element vector or as four individual parameters, and the elements or parameters may be integers or floats, and there are variations on the `glFog*()` function for each. The details of the individual versions of `glFog*()` are very similar to `glColor*()` and `glMaterial*()` and we refer you to the manuals for the details. Because fog is applied to graphics elements but not the background, it is a very good idea to make the fog and background colors be the same.

There are two additional options that we will skim over lightly, but that should at least be mentioned in passing. First, it is possible to use fog when you are using indexed color in place of RGB or RGBA color; in that case the color indices are interpolated instead of the color specification. (We did not cover indexed color when we talked about color models, but some older graphics systems only used this color technology and you might want to review that in your text or reference sources.) Second, fog is hintable — you may use `glHint(…)` with parameter `GL_FOG_HINT` and any of the hint levels to speed up rendering of the image with fog.

Fog is an easy process to illustrate. All of fog's effects can be defined in the initialization function, where the fog mode, color, density, and starting and ending points are defined. The actual imaging effect happens when the image is rendered, when the color of graphical elements are determined by blending the color of the element with the color of fog as determined by the fog mode. The various fog-related functions are shown in the code fragment below.

```
void myinit(void)
{     ...
      static  GLfloat fogColor[4]={0.5,0.5,0.5,1.0}; // 50% gray
      ...
// define the fog parameters
      glFogi(GL_FOG_MODE, GL_EXP);        // exponential fog increase
      glFogfv(GL_FOG_COLOR, fogColor);    // set the fog color
```

```
        glFogf(GL_FOG_START, 0.0 );          // standard start
        glFogf(GL_FOG_END, 1.0 );            // standard end
        glFogf(GL_FOG_DENSITY, 0.50);        // how dense is the fog?
        ...
        glEnable(GL_FOG);                    // enable the fog
...
}
```

An example illustrates our perennial cube in a foggy space, shown in Figure 13.6. The student is encouraged to experiment with the fog mode, color, density, and starting and ending values to examine the effect of these parameters' changes on your images. This example has three different kinds of sides (red, yellow, and texture-mapped) and a fog density of only 0.15, and has a distinctly *non*-foggy background for effect.

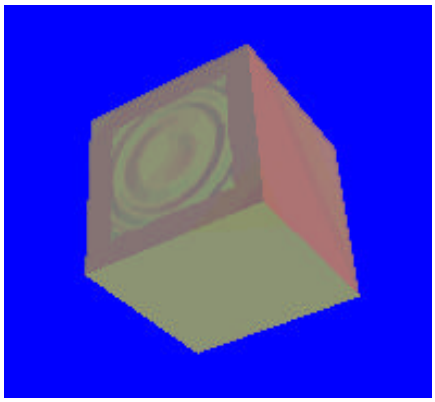


Figure 13.6: a foggy cube (including a texture map on one surface)

Fog is a tempting technique because it looks cool to have objects that aren't as sharp and "finished" looking as most objects seem to be in computer graphics. This is similar to the urge to use texture mapping to get objects that don't seem to be made of smooth plastic, and the urge to use smooth-shaded objects so they don't seem to be crudely faceted. In all these cases, though, using the extra techniques has a cost in extra rendering time and programming effort, and unless the technique is merited by the communication needed in the scene, it can detract from the real meaning of the graphics.

Collision detection

When you do polygon-based graphics, the question of collisions between objects reduces to the question of collisions between polygons. We discussed this earlier in Chapter 4, but will review that discussion here.

The first steps are to avoid doing any unnecessary work by testing first for situations where collisions are impossible. You can set up bounding volumes, for example, and determine that two bounding volumes cannot intersect. If an intersection is possible, however, then you reduce the problem by working with the actual objects, and usually work with the triangles that most commonly make up your objects. By reducing the general polygon to a triangle, that further reduces to the question of collisions between an edge and a triangle. We actually introduced this issue earlier in the mathematical background, but it boils down to extending the edge to a complete line, intersecting the line with the plane of the polygon, and then noting that the edge meets the polygon if it meets a sequence of successively more focused criteria:

- the parameter of the line where it intersects the plane must lie between 0 and 1

- the point where the line intersects the plane must lie within the smallest circle containing the triangle
- the point where the line intersects the plane must lie within the body of the triangle.

This comparison process is illustrated in Figure 13.7 below.

If you detect a collision when you are working with moving polyhedra, the presence of an intersection might require more processing because you want to find the exact moment when the moving polyhedra met. In order to find this intersection time, you must do some computations in the time interval between the previous step (before the intersection) and the current step (when the intersection exists). You might want to apply a bisection process on the time, for example, to determine whether the intersection existed or not halfway between the previous and current step, continuing that process until you get a sufficiently good estimate of the actual time the objects met. Taking a different approach, you might want to do some analytical computation to calculate the intersection time given the positions and velocities of the objects at the previous and current times so you can re-compute the positions of the objects to reflect a bounce or other kind of interaction between them.

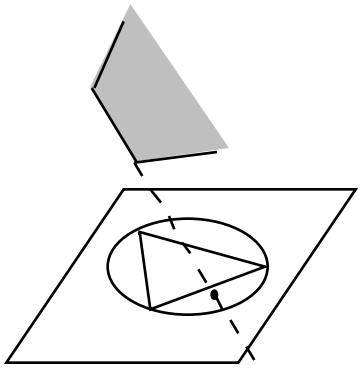


Figure 13.7: the collision detection computation

Chapter 14: Object Selection

Prerequisites

An understanding of the rendering process, an understanding of event handling, and a knowledge of list management to handle hit lists for events

Introduction

In the earlier chapter on interaction, we saw many ways that a graphics API would support user inputs into your images through menus, keystrokes, and mouse functions. If you wanted to identify one of the objects in the scene to act on, you could use those tools to identify an object by name, but you could not identify an object simply by clicking on it. In this chapter we will show you how you can make that kind of selection, or to use the term that's most common in graphics, we will show you how to *pick* an object in a scene. This kind of object picking permits the user to interact with a scene in a much more direct way than is possible with the kind of external events, such as menu selections, mouse clicks, or key presses, that we saw in the earlier chapter on event handling.

With object picking we can get the kind of direct manipulation that we are familiar with from graphical user interfaces, where the user selects a graphical object and then applies operations to it. Conceptually, picking allows your user to identify a particular object with the cursor and to choose it by clicking the mouse button when the cursor is on the object. The program must be able to identify what was selected, and then must have the ability to apply whatever action the user chooses to that particular selected object.

To understand how picking works, let's start with the mouse click. When you get a mouse event, the event callback gets four pieces of information: the button that was clicked, the state of that button, and the integer coordinates of the point in the window where the event happened. In order to find out what object might be indicated by the click, we convert the window coordinates to 2D eye coordinates but we must then reverse the projection and go back from 2D eye space to 3D eye space. However, a single point in 2D eye space becomes a line in 3D eye space, as shown in Figure 14.1. Our problem then becomes how to identify what objects meet this line segment and which of those objects was chosen by the user.

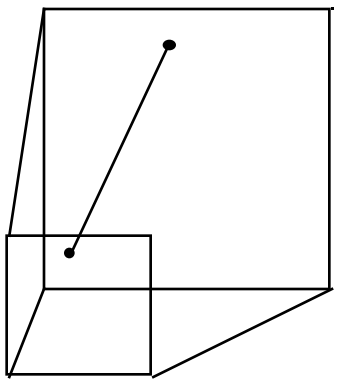


Figure 14.1: the line in the view volume that corresponds to a point in 2D eye space

It is possible to make this computation directly from the geometry using collision detection logic. That is, for each object in the scene, we calculate whether or not the line and object intersect. When we have done all these calculations, we can say which objects lie under the point that was chosen and where each of those intersections takes place in 3D eye space. We may then choose the

closest, which is be the one that the user would see at the place the mouse click was made, or we may choose any other intersection that our logic demands. The problem with this is that it is very computation intensive and requires us to be able to go back to eye space, and that makes it difficult to implement.

Another way to identify which objects lie on the line is to invert this logic completely. Instead of focusing on the objects that lie under the line of the pick, we see that any object that we might have chosen will use the pixel that was chosen when it was rendered. So if we can keep track of that pixel and save information on any object that includes it, we can identify all the objects that lie under the pick point. Because the rendering process keeps track of depth values, we can also get information on the depth of the object in the view volume when the pixel is used. Note that we do not talk about what kind of information is saved, so there are some interesting opportunities to think about what might be done here.

We cannot say which of these techniques might be used by any particular graphics application, and we cannot say which might be used by a particular graphics API. But OpenGL uses the second technique, and in the rest of this chapter we discuss how that is done.

Picking in OpenGL

OpenGL has several ways to identify objects that correspond to mouse events, and we will discuss two of them in this chapter. One of these involves drawing invisibly and keeping track of objects that include a given pixel or small region around a pixel; this is what we will call the standard selection approach. The other involves drawing with synthetic colors that are unique to each object and looking at the color of the selected pixel in the color buffer to identify the nearest object at the picked point. We will discuss the standard selection approach first.

The standard selection approach calls for the mouse event to request that you render your scene invisibly so a record may be made of all the objects that are selected. This approach introduces the concept of the *render mode* for drawing. In your standard rendering, you draw the scene in `GL_RENDER` mode, which is the default drawing mode. In the mouse event callback that is executed after the mouse event, you change your rendering to `GL_SELECT` mode and re-draw the scene with each item of interest given a unique name. When the scene is rendered in `GL_SELECT` mode, nothing is actually changed in the frame buffer but the pixels that would be rendered are identified. When any named object is found that would include the pixel selected by the mouse, that object's name is added to a selection buffer data structure, actually a stack of unsigned integers, that is maintained for that name. This selection buffer holds information on all the items in a hierarchy of named items that were hit. When the rendering of the scene in `GL_SELECT` modeis finished, a list of hit records is produced, with one entry for each name of an object whose rendering included the mouse click point, and the number of such records is returned when the system is returned to `GL_RENDER` mode. The structure of these hit records is described below. You can then process this list to identify the items that were hit, including the distance from the eye where the hit occurred, and you can proceed to do whatever work you need with this information.

The concept of "item of interest" is more complex than is immediately apparent. It can include a single object, a set of objects, or even a hierarchy of objects. Think creatively about your problem and you may be surprised just how powerful this kind of selection can be.

Definitions

The first concept we must deal with for object selection is the notion of a *selection buffer*. This is an array of unsigned integers (`GLuint`) that will hold the array of *hit records* for a mouse click. In turn, a hit record contains several items as illustrated in Figure 14.2. These include the number

of items that were on the *name stack*, the nearest (`zmin`) and farthest (`zmax`) distances to objects on the stack, and the list of names on the name stack for the selection. The distances are integers because they are taken from the *Z*-buffer, where you may recall that distances are stored as integers in order to make comparisons more effective. The name stack contains the names of all the objects in a hierarchy of named objects that were selected with the mouse click.

The distance to objects is given in terms of the viewing projection environment, in which the nearest points have the smallest non-negative values because this environment has the eye at the origin and distances increase as points move away from the eye. Typical processing will examine each selection record to find the record with the smallest value of `zmin` and will work with the names in that hit record to carry out the work needed by that hit. This work is fairly typical of the handling of any list of variable-length records, proceeding by accumulating the starting points of the individual records (starting with 0 and proceeding by adding the values of `(nitems+3)` from the individual records), with the `zmin` values being offset by 1 from this base and the list of names being offset by 3. This is not daunting, but it does require some care.

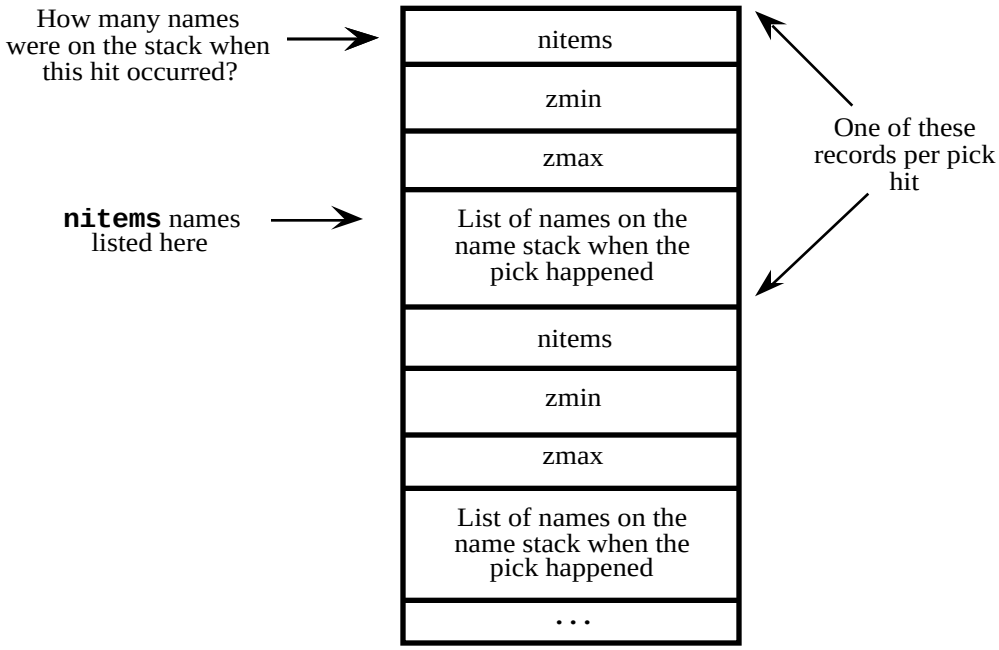


Figure 14.2: the structure of the selection buffer

In OpenGL, we have two ways to pick an object. We can generate the scene (partly or fully) and record the intersection of the object with the pixel identified by the mouse, or we can set up an additional projection with the pick matrix and clip everything that is not within a certain distance of the pixel we chose. The first of these methods is perhaps simpler because it does not involve any changes in the basic rendering, but the second is faster because it uses clipping to avoid rendering anything except those items very near the pixel. We will begin by discussing the simpler case, and describe the use of the pick matrix later.

Before we go on, however, we must be sure you understand one key point: you don't do anything more to generate the selection buffer than to draw the scene in select mode. That's it. The system does all the rest of the work for you, putting the active name (or all the names on the active name stack) into the selection buffer whenever the drawing process operates on the selected pixel or the object is not clipped away by the pick matrix. You simply generate the scene in select mode, get

the number of hits from the returned value of the `glRenderMode(...)` function when you call it to restore `GL_RENDER` mode, and process the selection buffer to handle that many hits.

Making picking work

The picking process is fairly straightforward. The function `glRenderMode(mode)` allows you to draw in either of two modes: render mode (`GL_RENDER`) invokes the graphics pipeline and produces pixels in the frame buffer, and select mode (`GL_SELECT`) calculates the pixels that would be drawn if the graphics pipeline were to be invoked, and tests the pixels against the pixels that were identified by the mouse click. As illustrated in the example below, the mouse function can be defined to change the drawing mode to `GL_SELECT` and to post a redisplay operation. The display function can then draw the scene in select mode with selection object names defined with `glutLoadName(int)` to determine what name will be put into the selection buffer if the object includes the selected pixel, noting that the mode can be checked to decide what is to be drawn and/or how it is to be drawn, and then the selection buffer can be examined to identify what was hit so the appropriate processing can be done. After the selection buffer is processed, the scene can be displayed again in render mode to present the effect of the selection.

In the outline above, it sounds as though the drawing in select mode will be the same as in render mode. But this is often not the case; there are some techniques to make the select mode drawing work more quickly or more flexibly than render-mode drawing. These include:

- if there are any objects that you don't want the user to be able to select, do not draw these at all in select mode. Because they are not drawn, they are invisible to picking.
- if you want to allow a pick of a complex object, you need not do all the work of a full rendering of the object in select mode; you need only design an approximation of the object and draw that.
- you can even create invisible controls by allowing the user to pick things that are only drawn in select mode but not in render mode.

Think creatively and you can find that you can do interesting things with selection. In fact, you will sometimes find that you must use some of these techniques. For example, if you want the user to be able to select a wireframe object, you probably want to replace the wireframe version by a solid version in the select mode drawing, because a user will visualize the spaces in the wireframe as part of the object but OpenGL will not. Another important use is in selecting text, because you cannot pick raster characters in OpenGL. For whatever reason, if you draw any raster characters in select mode, OpenGL will always think that the characters were picked no matter where you clicked. If you want to be able to pick a word that is drawn as raster characters, create a rectangle that occupies the space where the raster characters would be, and draw that rectangle in select mode.

It's worth a word on the notion of selection names. You cannot load a new name inside a `glBegin(mode)-glEnd()` pair, so if you use any geometry compression in your object, it must all be within a single named object. You can, however, nest names with the *name stack*, using the `glPushName(int)` function so that while the original name is active, the new name is also active. For example, suppose we were dealing with automobiles, and suppose that we wanted someone to select parts for an automobile. We could permit the user to select parts at a number of levels; for example, to select an entire automobile, the body of the automobile, or simply one of the tires. In the code below, we create a heirarchy of selections for an automobile ("Jaguar") and for various parts of the auto ("body", "tire", etc.) In this case, the names `JAGUAR`, `BODY`, `FRONT_LEFT_TIRE`, and `FRONT_RIGHT_TIRE` are symbolic names for integers that are defined elsewhere in the code.

```
glLoadName( JAGUAR );
glPushName( BODY );
     glCallList( JagBodyList );
```

```
glPopName();
glPushName( FRONT_LEFT_TIRE );
     glPushMatrix();
     glTranslatef( ??, ??, ?? );
     glCallList( TireList );
     glPopMatrix();
glPopName();
glPushName( FRONT_RIGHT_TIRE );
     glPushMatrix();
     glTranslatef( ??, ??, ?? );
     glCallList( TireList );
     glPopMatrix();
glPopName();
```

When a selection occurs, then, the selection buffer will include everything whose display involved the pixel that was chosen, including the automobile as well as the lower-level part. For example, if you selected the right front tire of the automobile, `nitems` would be 3 and your hit record would include three names: the names `FRONT_LEFT_TIRE`, `BODY`, and `JAGUAR`. Your program then would know that a hierarchy was selected that had these three parts and could choose (or allow the user to choose) which selection or what other logic it needed to use.

There are a couple of things to watch out for in the name stack. The first is that the name stack is empty when it is initialized, so you cannot simply load a name into the stack; this will generate an error. Instead you must push some name onto the stack so that you can load a name to replace it. The second thing to watch for with the name stack is that loading a name only replaces the top name on the name stack. If you have finished a hierarchy and need to remove the entire hierarchy from the name stack, you will need to pop the name stack until there is a single name left; you can then load the new name and replace that single name. But these are straightforward to remember as you use the name stack.

The pick matrix

Picking using the pick matrix is almost the same operation, logically, as picking using the selected pixel, but we present it separately because it uses a different process and allows us to define a concept of "near' and to talk about a way to identify the objects near the selection point.

In the picking process, you can define a very small window in the immediate neighborhood of the point where the mouse was clicked, and then you can identify everything that is drawn in that neighborhood. The result is returned in the same selection buffer and can be processed in the same way. This is done by creating a transformation with the function `gluPickMatrix(...)` that is applied after the projection transformation (that is, defined before the projection; recall the relation between the sequence in which transformations are identified and the sequence in which they are applied). The full function call is

```
gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height,
              GLint viewport[4])
```

where *x* and *y* are the coordinates of the point picked by the mouse, which is the center of the picking region; the width and height are the size of the picking region in pixels, sometimes called the pick tolerance; and the viewport is the vector of four integers returned by the function call `glGetIntegerv(GL_VIEWPORT, GLint *viewport)`.

The function of this pick matrix is to identify a small region centered at the point where the mouse was clicked and to select anything that is drawn in that region. The picking process returns a standard selection buffer that can then be processed to identify the objects that were picked, as described above.

A code fragment to implement this picking is given below. This corresponds to the point in the code for `doSelect(...)` above labeled "set up the standard viewing model" and "standard perspective viewing":

```
int viewport[4]; /* place to retrieve the viewport numbers */
   ...
dx = glutGet( GLUT_WINDOW_WIDTH );
dy = glutGet( GLUT_WINDOW_HEIGHT );
   ...
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
if( RenderMode == GL_SELECT ) {
   glGetIntegerv( GL_VIEWPORT, viewport );
   gluPickMatrix( (double)Xmouse, (double)(dy - Ymouse),
   PICK_TOL, PICK_TOL, viewport );
}
   ... call glOrtho(), glFrustum(), or gluPerspective() here
```

Using the back color buffer to do picking

There is another approach you can use that avoids the selection buffer entirely by using some of the facilities of double-buffered drawing. In this approach, when you want to permit a selection, you continue to draw your scene in render mode but you draw your scene into the back buffer in a unique way. Here you can identify the objects you want to make selectable by giving each a unique color, you can use a proxy for an object so that you draw the alternate representation we discussed above, and you can omit any objects you don't want to draw. When the mouse event happens and the mouse callback gets the pixel location of the pick, you look in the back buffer to see what color is at that pixel location. That color can be used to identify the object drawn at that position, giving you your picked object. If you were drawing with depth test enabled, you will get the object that is closest to the eye at that point. If you were not, you will get the last object drawn to that point. But you can control which of these you want to happen. And after you have gotten the information you need from the back buffer, you simply don't swap it with the front buffer but let the next (and probably normal) drawing operation replace the artificial-color image you have just created.

The mechanics of this are pretty straightforward. After the back buffer has been filled with the artificial image, select the back buffer to be read with the `glReadBuffer(GL_BACK)` function (although the back buffer is the default buffer for reading in double-buffered mode). Then use the `glReadPixels(...)` function to read a 1x1 array of color pixels (that is, the value of the color at a single point) at the position of the selection, and use whatever logic you created in determining the color of each object to identify the object you have just selected. This is a straightforward technique but it may require some thought to make a reasonable set of color identifications if you have a large number of objects.

A selection example

The selection process is pretty well illustrated by some code by a student, Ben Eadington. This code sets up and renders a Bézier spline surface with a set of selectable control points. When an individual control point is selected, that point can be moved and the surface responds to the adjusted set of points. An image from this work is given in Figure 14.3, with one control point selected (shown as being a red cube instead of the default green color).

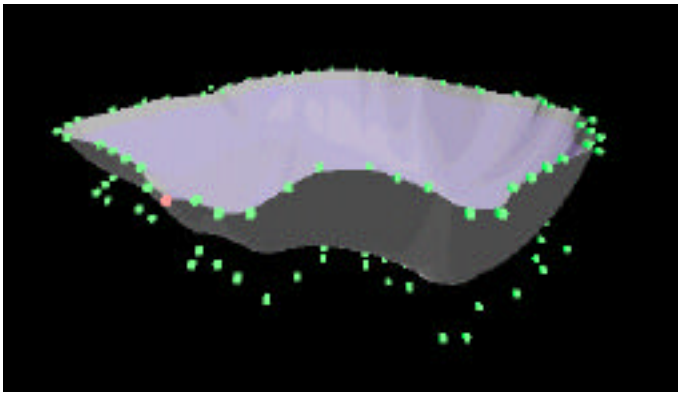


Figure 14.3: a surface with selectable control points and with one selected

Selected code fragments from this project are given below. Here all the data declarations and evaluator work are omitted, as are some standard parts of the functions that are presented, and just the important functions are given with the key points described in these notes. You will be directed to several specific points in the code to illustrate how selection works, described with interspersed text as the functions or code are presented.

In the first few lines you will see the declaration of the global selection buffer that will hold up to 200 values. This is quite large for the problem here, since there are no hierarchical models and no more than a very few control points could ever line up. The actual size would need to be no more than four `GLuints` per control point selected, and probably no more than 10 maximum points would ever line up in this problem. Each individual problem will need a similar analysis.

```
// globals initialization section
#define MAXHITS 200    // number of GLuints in hit records
// data structures for selection process
GLuint selectBuf[MAXHITS];
```

The next point is the mouse callback. This simply catches a mouse-button-down event and calls the DoSelect function, listed and discussed below, to handle the mouse selection. When the hit is handled (including the possibility that there was no hit with the cursor position) the control is passed back to the regular processes with a redisplay.

```
// mouse callback for selection
void Mouse(int button, int state, int mouseX, int mouseY)
{
   if (state == GLUT_DOWN) { // find which object was selected
      hit = DoSelect((GLint) mouseX, (GLint) mouseY);
   }
   glutPostRedisplay(); /* redraw display */
}
```

The control points may be drawn in either `GL_RENDER` or `GL_SELECT` mode, so this function must handle both cases. The only difference is that names must be loaded for each control point, and if any of the points had been hit previously, it must be identified so it can be drawn in red instead of in green. But there is nothing in this function that says what is or is not hit in another mouse click; this is handled in the DoSelect function below.

```
void drawpoints(GLenum mode)
{
   int i, j;
```

```
      int name=0;
      glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
      // iterate through control point array
      for(i=0; i<GRIDSIZE; i++)
         for(j=0; j<GRIDSIZE; j++) {
            if (mode == GL_SELECT) {
               glLoadName(name); // assign a name to each point
               name++;           // increment name number
            }
            glPushMatrix();
            ... place point in right place with right scaling
            if(hit==i*16+j%16) { // selected point, need to draw it red
               glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
               glutSolidCube(0.25);
               glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
            }
            else glutSolidCube(0.25);
            glPopMatrix();
         }
   }
```

The only real issue here is to decide what you do and do not need to draw in each of the two rendering modes. Note that the surface is only drawn if the program is in GL_RENDER mode; because nothing in the surface is itself selectable, the only thing that needs to be drawn in GL_SELECT mode is the control points.

```
   void render(GLenum mode) {
      ...  do appropriate transformations
      if (mode == GL_RENDER) { // don't render surface if mode is GL_SELECT
         surface(ctrlpts);
         ... some other operations that don't matter here
      }
      if(points) drawpoints(mode); // always render the control points
      ...  pop the transform stack as needed and exit gracefully
   }
```

This final function is the real meat of the problem. The display environment is set up (projection and viewing transformations), the `glRenderMode` function sets the rendering mode to `GL_SELECT` and the image is drawn in that mode, the number of hits is returned from the call to the `glRenderMode` function when it returns to `GL_RENDER` mode, the display environment is rebuilt for the next drawing, and the selection buffer is scanned to find the object with the smallest `zmin` value as the selected item. That value is then returned so that the `drawpoints` function will know which control point to display in red and so other functions will know which control point to adjust.

```
   GLint DoSelect(GLint x, GLint y)
   {
      int i;
      GLint hits, temphit;
      GLuint zval;

      glSelectBuffer(MAXHITS, selectBuf);
      glRenderMode(GL_SELECT);
      glInitNames();
      glPushName(0);

      // set up the viewing model
```

```
      ... standard perspective viewing and viewing transformation setup

      render(GL_SELECT);  // draw the scene for selection

      // find the number of hits recorded and reset mode of render
      hits = glRenderMode(GL_RENDER);
      // reset viewing model
      ... standard perspective viewing and viewing transformation setup
      // return the label of the object selected, if any
      if (hits <= 0) return -1;
      else {
         zval =  selectBuf[1];
         temphit = selectBuf[3];
         for (i = 1; i < hits; i++) { // for each hit
            if (selectBuf[4*i+1] < zval) {
               zval = selectBuf[4*i+1];
               temphit = selectBuf[4*i+3];
            }
         }
      }
      return temphit;
   }
```

A word to the wise...

This might be a good place to summarize the things we've seen about the standard picking process in the discussions and code examples above:

- Define an array of unsigned integers to act as the selection buffer
- Design a mouse event callback that calls a function that does the following:
  - Sets `GL_SELECT` mode and draws selected parts of the image, having loaded names so these parts can be identified when the selection is made
  - when this rendering is completed, returns a selection buffer that can be processed
  - returns to `GL_RENDER` mode.
- Be careful to manage your name stack as you draw objects in `GL_SELECT` mode so that you have exactly the things on the stack you need to identify the objects you want the user to be able to pick.

This design structure is straightforward to understand and can be easily implemented with a little care and planning.

Questions

1. Imagine, or lay out, some collection of objects in a space; an example might be the carousel model you created in Chapter 2. Suppose you only want part of these objects to be selectable (for example, the posts in the carousel, but not the carousel animals). Describe how you could define your selection process to make this happen.

2. Discuss the differences between picking and selection in terms of efficiency and of ease of identifying objects of various sizes. What advantages or disadvantages does each approach have?

3. Both selection and pick processes require you to analyze the selection buffer to identify which objects are closest, farthest, or have another relationship to the eye point. Why does the back buffer approach find the object nearest the eye without any further work? Is there any way that the back buffer could find any other object?

Exercises

4. Examine the nature of hit records by modifying any program including selection to include code to dump the selection buffer byte-by-byte into a file, and examining that file by a simple file dump utility such as Unix's `od`. Identify all the components of the selection buffer within this byte array and see how these components are really arranged.

5. Do the previous exercise when several objects are grouped in one name; when objects are arranged in a hierarchy. These should give more complex lists of the names on the name stack when the selection is made; break these down to understand how grouping and hierarchy work.

Experiments

6. Recall from Chapter 1 on projection that you calculated the parametric equation of a line segment in the viewing frustum that represents the points in 3D eye space that project to a single screen point. Define a number of simple sphere and polygon primitives that lie in the visible part of a space and choose a point on the front viewing plane. Calculate the intersection of the resulting line with each of the primitives and explore the way you could tell which is nearest the eye.

7. Experiment with the use of the back buffer for picking by setting your select-mode rendering to draw different objects in different colors and identify what object is nearest the eye at any given screen point.

Chapter 15: Interpolation and Spline Modeling

Prerequisites

A modest understanding of parametric functions of two variables together with an understanding of simple modeling with techniques such as triangle strips.

Introduction

In the discussions of mathematical fundamentals at the beginning of these notes, we talked about line segments as linear interpolations of points. Here we introduce other kinds of interpolations of points involving techniques called spline curves and surfaces. The specific spline techniques we will discuss are straightforward but we will limit them to one-dimensional spline curves. Extending these to two dimensions to model surfaces is a bit more complex and we will only cover this in terms of the evaluators that are built into the OpenGL graphics API. In general, spline techniques provide a very broad approach to creating smooth curves that approximate a number of points in a one-dimensional domain (1D interpolation) or smooth surfaces that approximate a number of points in a two-dimensional domain (2D interpolation). This interpolation is usually thought of as a way to develop geometric models, but there are a number of other uses of splines that we will mention later. Graphics APIs such as OpenGL usually provide tools that allow a graphics programmer to create spline interpolations given only the original set of points, called *control points*, that are to be interpolated.

In general, we think of an entire spline curve or spline surface as a single piece of geometry in the scene graph. These curves and surfaces are defined in a single modeling space and usually have a single set of appearance parameters, so in spite of their complexity they are naturally represented by a single shape node that is a leaf in the scene graph.

Interpolations

When we talked about the parametric form for a line segment in the early chapter on mathematical foundations for graphics, we created a correspondence between the unit line segment and an arbitrary line segment and were really interpolating between the two points by creating a line segment between them. If the points are named `P0` and `P1`, this interpolating line segment can be expressed in terms of the parametric form of the segment:

`(1-t)*P0 + t*P1`, for t in `[0.,1.]`

This form is almost trivial to use, and yet it is quite suggestive, because it hints that the set of points that interpolate the two given points can be computed by an expression such as

f_0`(t)*P0 +` f_1`(t)*P1`

for two fixed functions f_0 and f_1. This suggests a relationship between points and functions that interpolate them that would allow us to consider the nature of the functions and the kind of interpolations they provide. In this example, we have f_0`(t)=(1-t)` and f_1`(t)=t`, and there are interesting properties of these functions. We see that f_0`(0)=1` and f_1`(0)=0`, so at `t=0`, the interpolant value is `P0`, while f_0`(1)=0` and f_1`(1)=1`, so at `t=1`, the interpolant value is `P1`. This tells us that the interpolation starts at `P0` and ends at `P1`, which we had already found to be a useful property for the interpolating line segment. Note that because each of the interpolating functions is linear in the parameter `t`, the set of interpolating points forms a line.

As we move beyond line segments that interpolate two points, we want to use the term interpolation to mean determining a set of points that approximate the space between a set of given points in the order the points are given. This set of points can include three points, four points, or even more. We assume throughout this discussion that the points are in 3-space, so we will be

creating interpolating curves (and later on, interpolating surfaces) in three dimensions. If you want to do two-dimensional interpolations, simply ignore one of the three coordinates.

Finding a way to interpolate three points `P0`, `P1`, and `P2` is more interesting than interpolating only two points, because one can imagine many ways to do this. However, extending the concept of the parametric line we could consider a quadratic interpolation in t as:

$(1-t)^2*P0 + 2t*(1-t)*P1 + t^2*P2$, for t in `[0.,1.]`

Here we have three functions f_0, f_1, and f_2 that participate in the interpolation, with $f_0(t)=(1-t)^2$, $f_1(t)=2t*(1-t)$, and $f_2(t)=t^2$. These functions have by now achieved enough importance in our thinking that we will give them a name, and call them the *basis functions* for the interpolation. Further, we will call the points `P0`, `P1`, and `P2` the *control points* for the interpolation (although the formal literature on spline curves calls them *knots* and calls the endpoints of an interpolation *joints*). This particular set of functions have a similar property to the linear basis functions above, with $f_0(0)=1$, $f_1(0)=0$, and $f_2(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, and $f_2(1)=1$, giving us a smooth quadratic interpolating function in t that has value `P0` if `t=0` and value `P1` if `t=1`, and that is a linear combination of the three points if `t=.5`. The shape of this interpolating curve is shown in Figure 15.1.

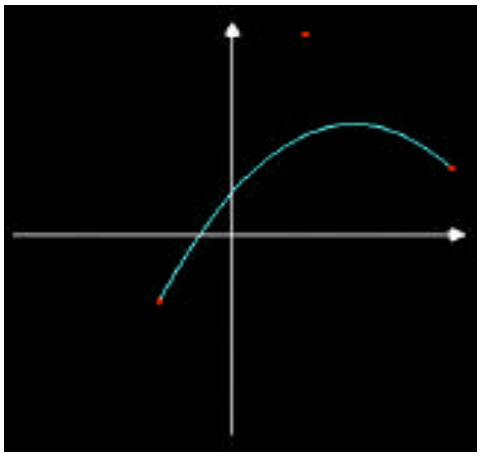


Figure 15.1: a quadratic interpolating curve for three points

The basis functions for these two cases are also of interest because they share a common source. In the case of linear interpolation, we can get the functions $f_0(t)$ and $f_1(t)$ as the terms when we expand $((1-t) + t)^1$, and in the case of quadratic interpolation, we can get the functions $f_0(t)$, $f_1(t)$, and $f_2(t)$ as the terms when we expand $((1-t) + t)^2$. In both cases, the functions all add to a value of 1 for any value of *t*, so we have an interpolation that gives a total weight of 1 for all the coefficients of the geometric points that we are using.

The observation above for the linear and quadratic cases is suggestive of a general approach. In this approach, we would use components which are products of the polynomials $((1-t) + t)^N$ and take their coefficients from the geometry we want to interpolate. If we follow this pattern for the case of cubic interpolations, interpolating four points `P0`, `P1`, `P2`, and `P3` would look like:

$(1-t)^3*P0 + 3t*(1-t)^2*P1 + 3t^2*(1-t)*P2 + t^3*P3$, for t in `[0.,1.]`

The shape of the curve this determines is illustrated in Figure 15.2. (We have chosen the first three of these points to be the same as the three points in the quadratic spline above to make it easier to compare the shapes of the curves). In fact, this curve is an expression of the standard Bézier spline function to interpolate four control points, and the four polynomials

$f_0(t)=(1-t)^3$,

$f_1(t)=3t(1-t)^2$,

$f_2(t)=3t^2(1-t)$, and

$f_3(t)=t^3$

are called the cubic *Bernstein basis* for the spline curve.

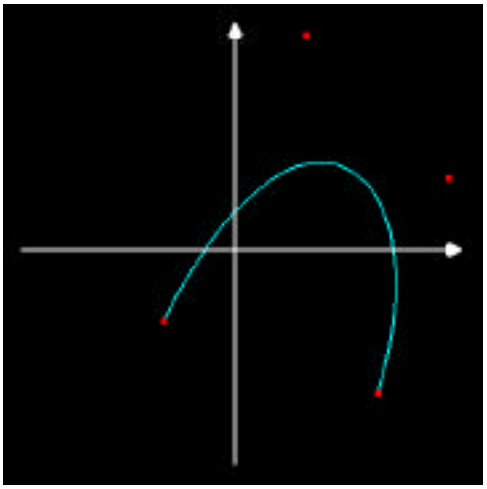


Figure 15.2: interpolating four points with the Bézier spline based on the Bernstein basis functions

When you consider this interpolation, you will note that the interpolating curve goes through the first and last control points (`P0` and `P3`) but does not go through the other two control points. This is because the set of basis functions for this curve behaves the same at the points where `t=0` and `t=1` as we saw in the quadratic spline: $f_0(0)=1$, $f_1(0)=0$, $f_2(0)=0$, and $f_3(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, $f_2(1)=0$, and $f_3(1)=1$. You will also note that as the curve goes through the first and last control points, it is moving in the direction from the first to the second control point, and from the third to the fourth control points. Thus the two control points that are not met control the shape of the curve by determining the initial and the ending directions of the curve, and the rest of the shape is determined in order to get the necessary smoothness.

This approach to getting the basis functions is not the only way to derive appropriate sets of functions. In general, curves that interpolate a given set of points need not go through those points, but the points influence and determine the nature of the curve in other ways. If you need to have the curve actually go through the control points, however, there are spline formulations for which this does happen. The Catmull-Rom cubic spline has the form

$f_0(t)$`*P0 +` $f_1(t)$`*P1 +` $f_2(t)$`*P2 +` $f_3(t)$`*P3`, for t in `[0.,1.]`

for basis functions

$f_0(t)=(-t^3+2t^2-t)/2$

$f_1(t)=(3t^3-5t^2+2)/2$

$f_2(t)=(-3t^3+4t^2+t)/2$

$f_3(t)=(t^3-t^2)/2$

This interpolating curve has a very different behavior from that of the Bézier curve above, because as shown in Figure 15.3. This is a different kind of interpolating behavior that is the result of a set of basis functions that have $f_0(0)=0$, $f_1(0)=1$, $f_2(0)=0$, and $f_3(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, $f_2(1)=1$, and $f_3(1)=0$. This means that the curve interpolates the points `P1` and `P2` instead of `P0` and `P3` and actually goes through those two points. Thus the Catmull-Rom spline curve is useful when you want your interpolated curve to include all the control points, not just some of them.

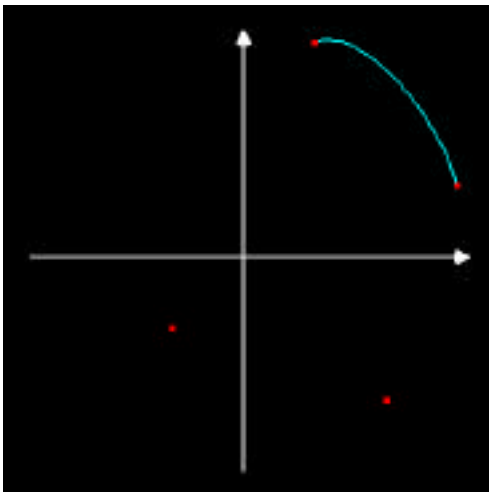


Figure 15.3: interpolating four points with the Catmull-Rom cubic spline

We will not carry the idea of spline curves beyond cubic interpolations, but we want to provide this much detailed background because it can sometimes be handy to manage cubic spline curves ourselves, even though OpenGL provides evaluators that can make spline computations easier and more efficient. Note that if the points we are interpolating lie in 3D space, each of these techniques provides a 3D curve, that is, a function from a line segment to 3D space.

Extending interpolations to more control points

While we have only shown the effect of these interpolations in the smallest possible set of points, it is straightforward to extend the interpolations to larger sets of points. The way we do this will depend on the kind of interpolation that is provided by the particular curve we are working with, however.

In the Bézier curve, we see that the curve meets the first and last control points but not the two intermediate control points. If we simply use the first four control points, then the next three (the last point of the original set plus the next three control points), and so on, then we will have a curve that is continuous, goes through every third control point (first, fourth, seventh, and so on), but that changes direction abruptly at each of the control points it meets. In order to extend these curves so that they progress smoothly along their entire length, we will need to add new control points that maintain the property that the direction into the last control point of a set is the same as the direction out of the first control point of the next set. In order to do this, we need to define new control points between each pair of points whose index is $2N$ and $2N+1$ for $N \geq 1$ up to, but not including, the last pair of control points. We can define these new control points as the midpoint

between these points, or $(P_{2N}+P_{2N+1})/2$. When we do, we get the following relation between the new and the original control point set:

original:	P0	P1	P2		P3	P4		P5	P6	P7
new:	P0	P1	P2	Q0	P3	P4	Q1	P5	P6	P7

where each point Q represents a new point calculated as an average of the two on each side of it, as above. Then the computations would use the following sequences of points: P0–P1–P2–Q0; Q0–P3–P4–Q1; and Q1–P5–P6–P7. Note that we must have an even number of control points for a Bézier curve, that we only need to extend the original control points if we have at least six control points, and that we always have three of the original points participating in each of the first and last segments of the curve.

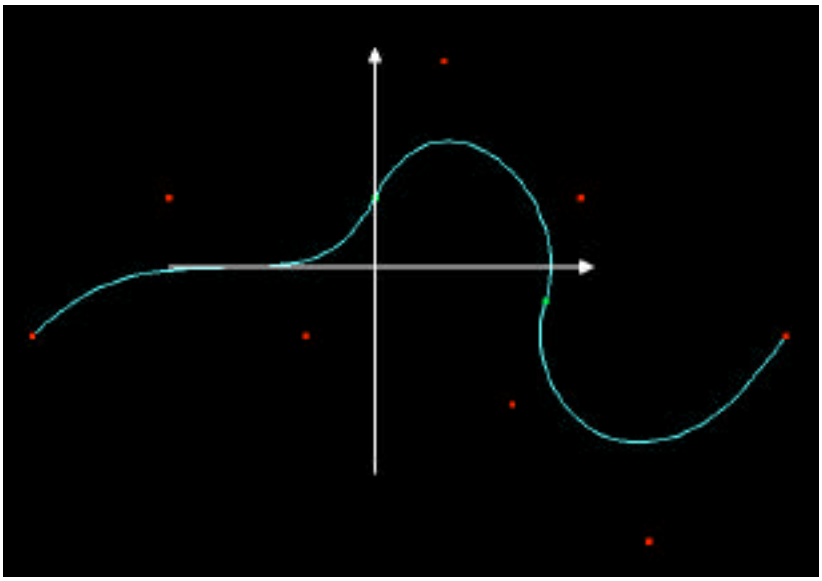


Figure 15.4: extending the Bézier curve by adding intermediate control points, shown in green

For the Catmull-Rom cubic spline, the fact that the interpolating curve only connects the control points P1 and P2 gives us a different kind of approach to extending the curve. However, it also gives us a challenge in starting the curve, because neither the starting control point P0 nor the ending control point P3 is not included in the curve that interpolates P0-P3. Hence we will need to think of the overall interpolation problem in three parts: the first segment, the intermediate segments, and the last segment.

For the first segment, the answer is simple: repeat the starting point twice. This gives us a first set of control points consisting of P0, P0, P1, and P2, and the first piece of the curve will then interpolate P0 and P1 as the middle points of these four. In the same way, to end the curve we would repeat the ending point, giving us the four control points P1, P2, P3, and P3, so the curve would interpolate the middle points, P2 and P3. If we only consider the first four control points and add this technique, we see the three-segment interpolation of the points shown in the left-hand image of Figure 15.5.

If we have a larger set of control points, and if we wish to extend the curve to cover the total set of points, we can consider a "sliding set" of control points that starts with P0, P1, P2, and P3 and, as we move along, includes the last three control points from the previous segment as the first three of the next set and adds the next control point as the last point of the set of four points. That is, the second set of points would be P1, P2, P3, and P4, and the one after that P2, P3, P4, and P5,

and so on. This kind of sliding set is simple to implement (just take an array of four points, move each one down by one index so `P[1]` becomes `P[0]`, `P[2]` becomes `P[1]`, `P[1]` becomes `P[1]`, and the new point becomes `P[3]`. The sequence of points used for the individual segments of the curve are then `P0–P0–P1–P2`; `P0–P1–P2–P3`; `P1–P2–P3–P4`; `P2–P3–P4–P5`; `P3–P4–P5–P6`; `P4–P5–P6–P7`; `P5–P6–P7–P8`; and `P6–P7–P8–P8`. The curve that results when we extend the computation across a larger set of control points is shown as the right-hand image of Figure 15.5, where we have taken the same setof control points that we used for the extended Bézier spline example.

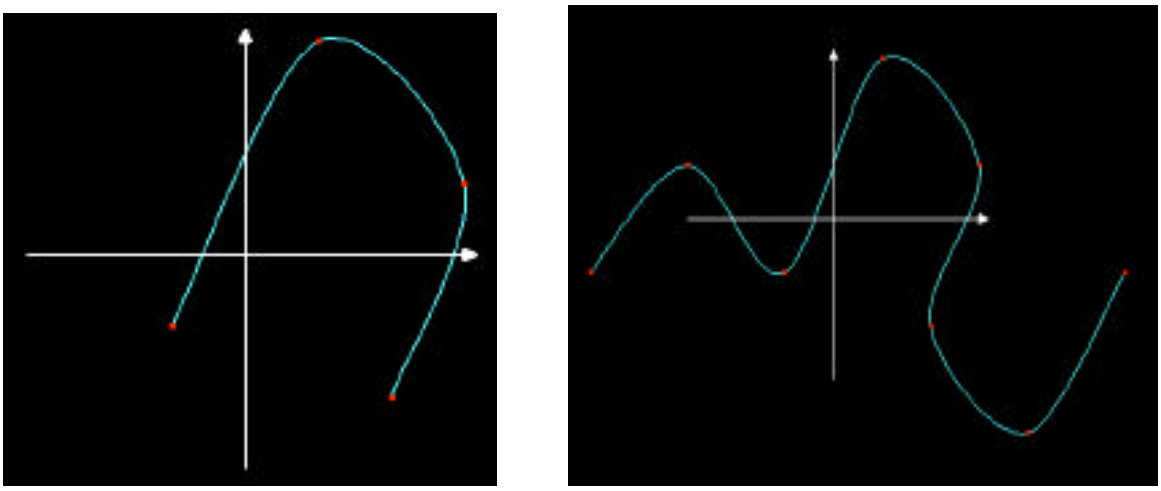


Figure 15.5: extending the Catmull-Rom curve by including the endpoints of the set (left) and by stepping along the extended set of control points (right)

The techniques defined above can easily be extended to generate interpolated surfaces. If you consider functions $f_0(t)$, $f_1(t)$, $f_2(t)$, and $f_3(t)$ as the basis for your cubic spline, you can apply them for two parameters, *u* and *v*, along with an extended set of 16 control points P_{ij} for *i* and *j* between 0 and 3, to give you a function of two variables:

$$f(u,v) = \sum(f_i(u)*f_j(v)*Pij)$$

where the sum is taken over the 16 possible values of *i* and *j*. You can then step along the two variables *i* and *j* in any way you like and draw the surface in exactly the same way you would graph a function of two variables, as we discussed in a much earlier chapter.

Interpolations in OpenGL

In OpenGL, the spline capability is provided by techniques called *evaluators*, functions that take a set of control points and produce another set of points that interpolate the original control points. This allows you to model curves and surfaces by doing only the work to set the control points and set up the evaluator, and then to get much more detailed curves and surfaces as a result. There is an excellent example of spline surfaces in the Eadington example for selecting and manipulating control points in the chapter of these notes on object selection.

There are two kinds of evaluators available to you. If you want to interpolate points to produce one-parameter information (that is, curves or any other data with only one degree of freedom; think 1D textures as well as geometric curves), you can use 1D evaluators. If you want to interpolate points in a 2D array to produce two-parameter information (that is, surfaces or any other data with two degrees of freedom; think 2D textures as well as geometric curves) you can use 2D evaluators.

Both are straightforward and allow you to choose how much detail you want in the actual display of the information.

In Figures 15.6 through 15.8 below we see several images that illustrate the use of evaluators to define geometry in OpenGL. Figure 15.6 shows two views of a 1D evaluator that is used to define a curve in space showing the set of 30 control points as well as additional computed control points for smoothness; Figure 15.7 shows a 2D evaluator used to define a single surface patch based on a 4x4 set of control points; and Figure 15.8 shows a surface defined by a 16x16 set of control points with additional intermediate control points not shown. These images and the techniques for creating smooth curves will be discussed further below, and some of the code that creates these is given in the Examples section.

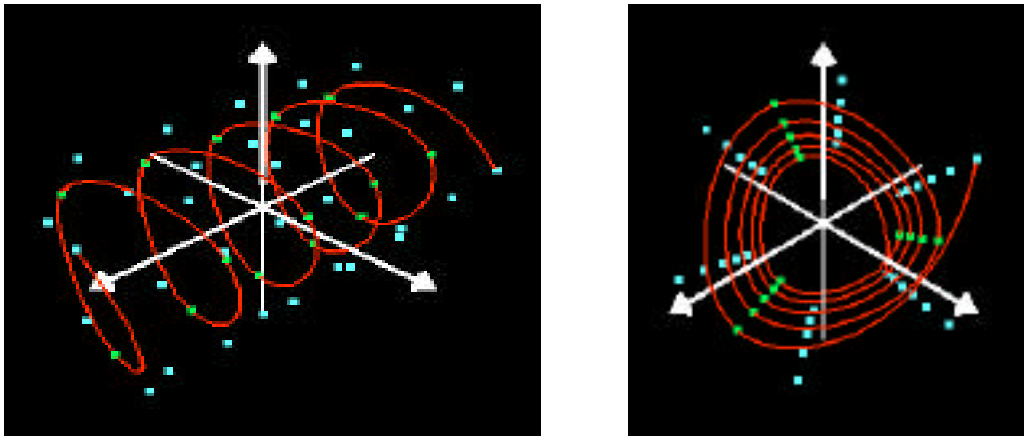


Figure 15.6: a spline curve defined via a 1D evaluator, shown from a point of view with x = y = z (left) and rotated to show the relationship between control points and the curve shape (right) The cyan control points are the originals; the green control points are added as discussed above.

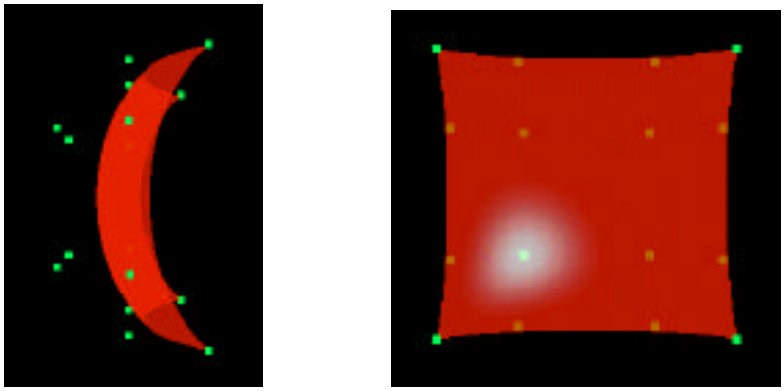


Figure 15.7: a spline patch defined by four control points using a 2D evaluator

The spline surface in Figure 15.7 has only a 0.7 alpha value so the control points and other parts of the surface can be seen behind the primary surface of the patch. In this example, note the relation between the control points and the actual surface; only the four corner points actually meet the surface, while all the others lie off the surface and act only to influence the shape of the patch. Note also that the entire patch lies within the convex hull of the control points. The specular highlight on the patch should also help you see the shape of the patch from the lighting. In the larger surface of Figure 15.8, note how the surface extends smoothly between the different sets of control points.

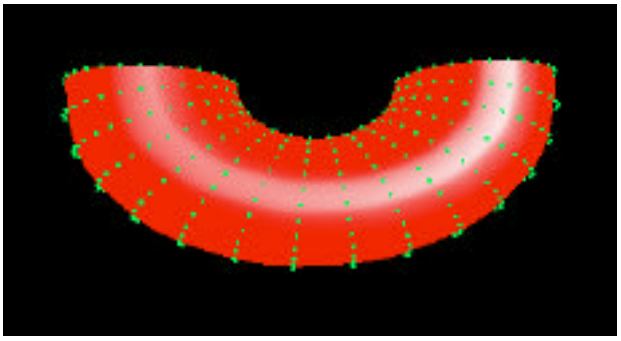


Figure 15.8: a spline surface defined by a 2D evaluator using a 16x16 set of control points. The original set of control points is extended with interpolated points as defined below

While these examples have used the full curve or surface generation capabilities of evaluators; this generates the entire mesh based on the control points. However, there are times when you might want to generate only a single point on a curve or a surface that corresponds to particular values of the parameter or parameters. For example, if you have developed a surface and you want to show a "fly-over" of the surface, you may want to position the eye along a curve that you define with control points. Then to generate images with eye points along this curve, you will want to define the eye position with particular values of the parameter that defines the curve. For each of these eye positions, you will want to evaluate the curve at the corresponding parameter value, getting the coordinates of that point to use with the `gluLookAt(...)` function.

Automatic normal and texture generation with evaluators

The images in Figure 15.7 and 15.8 include the standard lighting model and show specular highlighting, but as you will see from the code example for spline surfaces later in this chapter, neither explicit vertex definitions nor explicit normal definitions are used. Instead, 2D evaluators are used to generate the vertices and vertex normals are generated automatically. Basically, the key elements are as follows:

- to specify an array of 16 control points,
- to enable `GL_MAP2_VERTEX_3` to specify that you will generate points in 3D space with a 2D map,
- to enable `GL_AUTO_NORMAL` to specify that normals are to be generated analytically from the evaluator,
- to use `glMap2f(…)` to specify the details of the 2D mapping and to identify that the control point array above is to be used,
- to use `glMapGrid2f(…)` to specify how the 2D domain space is to be used in creating the vertex grid to be displayed, and
- to use `glEvalMesh2(…)` to carry out the evaluation and display the surface or, if you only want to calculate a point on the surface that corresponds to a single parameter pair (u,v), use `glEvalCoord2f(u,v)` to get the coordinates of that point.

There are, in fact, a number of similar functions that you can use for similar operations in different dimensions, and you should look these up in an OpenGL manual for more details.

Besides generating automatic normals, there are other capabilities for generating information for your evaluator surface. You can generate automatic texture coordinates for your evaluator surfaces. If you enable `GL_MAP2_TEXTURE_COORD_2`, for example, and use `glMap2f(…)` with the first parameter `GL_MAP2_TEXTURE_COORD_2` and with similar parameters to those you used for vertex generation, then the `glEvalMesh2(…)` function will generate s and t coordinates

for the texture on your 2D patch. As above, there are many variations to generate 1D through 4D textures from 1D or 2D meshes; see the OpenGL manuals for details. The image from Figure 15.7 is extended to Figure 15.9 by changing the color to white and applying the texture with modulation with automatic texture coordinates. Sample code for Figures 15.7 and 15.9 that includes automatic normal and texture coordinate generation is included later in this chapter.

Figure 15.9: a texture map on the patch of Figure 15.7 created with automatic texture coordinates

You can also generate the normals from your surface from the `glMap2f(...)` function if you use the `GL_MAP2_NORMAL_4` parameter, or the colors from your surface if you use the `GL_MAP2_COLOR_4` parameter.

Additional techniques

Spline techniques may also be used for much more than simply modeling. Using them, you can generate smoothly changing sets of colors, or of normals, or of texture coordinates—or probably just about any other kind of data that one could interpolate. There aren't built-in functions that allow you to apply these points automatically as there are for creating curves and surfaces, however. For these you will need to manage the parametric functions yourself. To do this, you need to define each point in the `(u,v)` parameter space for which you need a value and get the actual interpolated points from the evaluator using the functions `glEvalCoord1f(u)` or `glEvalCoord2f(u,v)`, and then use these points in the same way you would use any points you had defined in another way. These points, then, may represent colors, or normals, or texture coordinates, depending on what you need to create your image.

To be more concrete, suppose you have a surface defined by two parameters, each having values lying in [0,1]. If you wanted to impose a set of normals on the surface to achieve a lighting effect, you could define a set of control points that approximated the normals you want. (Recall that a normal is just a 3D vector, just as is a point.) You can then define an evaluator for your new surface defined by the control points and a mapping from the parameters of the original surface to the parameters on the new surface. To define a normal at a point with parametric coordinates (u,v), then, you would determine the corresponding parameters (u',v') on the new surface, get the value (x,y,z) of the original surface as $f(u,v)$ for whatever parametric function you are using and get the value (r,s,t) of the new surface with the `glEval2f(u',v')` function, and then use these values for the vertex in the function calls

```
glNormal3f(r,s,t); glVertex3f(x,y,z);
```

that would be used in defining the geometry for your image.

Another common example of spline use is in animation, where you can get a smooth curve for your eyepoint to follow by using splines. As your eyepoint moves, however, you also need to deal with the other issues in defining a view. The up vector is fairly straightforward; for simple animations, it is probably enough to keep the up vector constant. The center of view is more of a challenge, however, because it has to move to keep the motion realistic. The suggested approach is to keep three points from the spline curve: the previous point, the current point, and the next point, and to use the previous and next points to set the direction of view; the viewpoint is then a point at a fixed distance from the current point in the direction set by the previous and next points. This should provide a reasonably good motion and viewing setup. Other applications of splines in animation include positioning parts of a model to get smooth motion.

Definitions

As you see in Figures 15.6 and 15.7, an OpenGL evaluator working on an array of four control points (1D) or 4x4 control points (2D) actually fits the extreme points of the control point set but does not go through any of the other points. As the evaluator comes up to these extreme control points, the tangent to the curve becomes parallel to the line segment from the extreme point to the adjacent control point, as shown in Figure 15.10 below, and the speed with which this happens is determined by the distance between the extreme and adjacent control points.

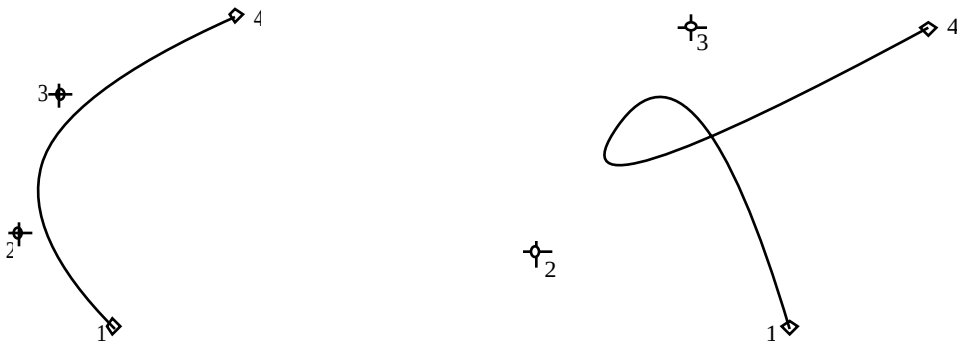


Figure 15.10: two spline curves that illustrate the shape of the curve with different layouts of control points

To control the shape of an extended spline curve, you need to arrange the control points so that the direction and distance from a control point to the adjacent control points are the same. This can be accomplished by adding new control points between appropriate pairs of the original control points as indicated in the spline curve figure above. This will move the curve from the first extreme point to the first added point, from the first added point smoothly to the second added point, from the second added point smoothly to the third added point, and so on to moving smoothly through the last added point to the last extreme point.

This construction and relationship is indicated by the green (added) control points in the first figure in this section. Review that figure and note again how there is one added point after each two original points, excepting the first and last points; that the added points bisect the line segment between the two points they interpolate; and that the curve actually only meets the added points, not the original points, again excepting the two end points. If we were to define an interactive program to allow a user to manipulate control points, we would only give the user access to the original control points; the added points are not part of the definition but only of the implementation of a smooth surface.

Similarly, one can define added control points in the control mesh for a 2D evaluator, creating a richer set of patches with the transition from one patch to another following the same principle of

equal length and same direction in the line segments coming to the edge of one patch and going from the edge of the other. This allows you to achieve a surface that moves smoothly from one patch to the next. Key points of this code are included in the example section below, but it does take some effort to manage all the cases that depend on the location of a particular patch in the surface. The example code below will show you these details.

So how does this all work? A cubic spline curve is determined by a cubic polynomial in a parametric variable *u* as indicated by the left-hand equation in (1) below, with the single parameter *u* taking values between `0` and `1`. The four coefficients a_i can be determined by knowing four constraints on the curve. These are provided by the four control points needed to determine a single segment of a cubic spline curve. We saw ways that these four values could be represented in terms of the values of four basis polynomials, and an OpenGL 1D evaluator computes those four coefficients based on the Bézier curve definition and, as needed, evaluates the resulting polynomial to generate a point on the curve or the curve itself. A bicubic spline surface is determined by a bicubic polynomial in parametric variables *u* and *v* as indicated by the right-hand equation in (1) below, with both parameters taking values between 0 and 1. This requires computing the 16 coefficients a_{ij} which can be done by using the 16 control points that define a single bicubic spline patch. Again, an OpenGL 2D evaluator takes the control points, determines those 16 coefficients based on the basis functions from the Bézier process, and evaluates the function as you specify to create your surface model.

(1)	$\sum_{i=0}^{3} a_i u^i$	$\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} u^i v^j$

Some examples

Spline curves:

The setup to generate curves is given in some detail below. This involves defining a set of control points for the evaluator to use, enabling the evaluator for your target data type, defining overall control points for the curve, stepping through the overall control points to build four-tuples of segment control points, and then invoking the evaluator to draw the actual curve. This code produced the figures shown in the figure above on spline curves. A few details have been omitted in the code below, but the essential parts of setting up the control points are fully included. Note that this code returns the points on the curve using the `glEvalCoord1f(...)` function instead of the `glVertex*(...)` function within a `glBegin(...)` … `glEnd()` pair; this is different from the more automatic approach of the 2D patch example that follows it.

Probably the key point in this sample code is the way the four-tuples of segment control points have been managed. The original points would not have given smooth curves, so as discussed above, new points were defined that interpolated some of the original points to make the transition from one segment to the other continuous and smooth.

```
   glEnable(GL_MAP1_VERTEX_3)

void makeCurve( void )
{
...
   for (i=0; i<CURVE_SIZE; i++) {
      ctrlpts[i][0]= RAD*cos(INITANGLE + i*STEPANGLE);
      ctrlpts[i][1]= RAD*sin(INITANGLE + i*STEPANGLE);
      ctrlpts[i][2]= -4.0 + i * 0.25;
```

```
   }
}

void curve(void) {
#define LAST_STEP (CURVE_SIZE/2)-1
#define NPTS 30

   int step, i, j;

   makeCurve(); // calculate the control points for the entire curve
   // copy/compute points from  ctrlpts  to  segpts to define each segment
   // of the curve.  First/last cases are different from middle cases...
   for ( step = 0; step < LAST_STEP; step++ ) {
      if (step==0) { // first case
         for (j=0; j<3; j++) {
            segpts[0][j]=ctrlpts[0][j];
            segpts[1][j]=ctrlpts[1][j];
            segpts[2][j]=ctrlpts[2][j];
            segpts[3][j]=(ctrlpts[2][j]+ctrlpts[3][j])/2.0;
         }
      }
      else if (step==LAST_STEP-1) { // last case
         for (j=0; j<3; j++) {
            segpts[0][j]=(ctrlpts[CURVE_SIZE-4][j]
                          +ctrlpts[CURVE_SIZE-3][j])/2.0;
            segpts[1][j]=ctrlpts[CURVE_SIZE-3][j];
            segpts[2][j]=ctrlpts[CURVE_SIZE-2][j];
            segpts[3][j]=ctrlpts[CURVE_SIZE-1][j];
         }
      }
      else for (j=0; j<3; j++) { // general case
         segpts[0][j]=(ctrlpts[2*step][j]+ctrlpts[2*step+1][j])/2.0;
         segpts[1][j]=ctrlpts[2*step+1][j];
         segpts[2][j]=ctrlpts[2*step+2][j];
         segpts[3][j]=(ctrlpts[2*step+2][j]+ctrlpts[2*step+3][j])/2.0;
      }

      // define the evaluator
      glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &segpts[0][0]);
      glBegin(GL_LINE_STRIP);
      for (i=0; i<=NPTS; i++)
         glEvalCoord1f( (GLfloat)i/(GLfloat)NPTS );
      glEnd();
      ...
   }
}
```

Spline surfaces:

We have two examples, the first showing drawing a simple patch (surface based on a 4x4 grid of control points) and the second showing drawing of a larger surface with more control points. Below is some simple code to generate a surface given a 4x4 array of points for a single patch, as shown in Figure 15.7 above. This code initializes a 4x4 array of points, enables auto normals (available through the `glEvalMesh(...)` function) and identifies the target of the evaluator, and carries out the evaluator operations. The data for the patch control points is deliberately over-simplified so you can see this easily, but in general the patch points act in a parametric way that is quite distinct from the indices, as is shown in the general surface code. This code also includes the

`glEnable(…)` and `glMapGrid2f(GL_MAP2_TEXTURE_COORD_2,...)` statements that are used to generate automatic texture coordinates for Figure 15.9, but does not include the details of the texture mapping code. Note that the third parameter of the `glMapGrid2f(...)` function that specifies the texture coordinate generation is 4.0; this corresponds to mapping the texture coordinates over four grid points.

```
point3 patch[4][4] = {{{-2.,-2.,0.},{-2.,-1.,1.},{-2.,1.,1.},{-2.,2.,0.}},
                      {{-1.,-2.,1.},{-1.,-1.,2.},{-1.,1.,2.},{-1.,2.,1.}},
                      {{1.,-2.,1.},{1.,-1.,2.},{1.,1.,2.},{1.,2.,1.}},
                      {{2.,-2.,0.},{2.,-1.,1.},{2.,1.,1.},{2.,2.,0.}}};

void myinit(void)
{
   ...
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_MAP2_TEXTURE_COORD_2);
   glEnable(GL_MAP2_VERTEX_3);
}

void doPatch(void)
{
// draws a patch defined by a 4 x 4 array of points
#define NUM 20 //

   glMaterialfv(...);  // whatever material definitions are needed

   glMap2f(GL_MAP2_VERTEX_3,0.0,1.0,3,4,0.0,1.0,12,4,&patch[0][0][0]);
   glMap2f(GL_MAP2_TEXTURE_COORD_2,0.0,4.0,3,4,0.0,4.0,12,4,,&patch[0][0][0]);
   glMapGrid2f(NUM, 0.0, 1.0, NUM, 0.0, 1.0);
   glEvalMesh2(GL_FILL, 0, NUM, 0, NUM);
}
```

The considerations for creating a complete surface with a 2D evaluator is similar to that for creating a curve with a 1D evaluator. You need to create a set of control points, to define and enable an appropriate 2D evaluator, to generate patches from the control points, and to draw the individual patches. These are covered in the sample code below.

The sample code below has two parts. The first is a function that generates a 2D set of control points procedurally; this differs from the manual definition of the points in the patch example above or in the pool example of the selection section. This kind of procedural control point generation is a useful tool for procedural surface generation. The second is a fragment from the section of code that generates a patch from the control points, illustrating how the new intermediate points between control points are built. Note that these intermediate points all have indices 0 or 3 for their locations in the patch array because they are the boundary points in the patch; the interior points are always the original control points. Drawing the actual patch is handled by the function `doPatch(...)` above in just the same way as it is handled for the patch example, so it is omitted here.

```
point3 ctrlpts[GRIDSIZE][GRIDSIZE];

void genPoints(void)
{
#define PI 3.14159
#define R1 6.0
#define R2 3.0
   int i, j;
```

```
   GLfloat alpha, beta, step;

   alpha = -PI;
   step = PI/(GLfloat)(GRIDSIZE-1);
   for (i=0; i<GRIDSIZE; i++) {
      beta = -PI;
      for (j=0; j<GRIDSIZE; j++) {
         ctrlpts[i][j][0] = (R1 + R2*cos(beta))*cos(alpha);
         ctrlpts[i][j][1] = (R1 + R2*cos(beta))*sin(alpha);
         ctrlpts[i][j][2] = R2*sin(beta);
         beta -= step;
      }
      alpha += step;
   }
}

void surface(point3 ctrlpts[GRIDSIZE][GRIDSIZE])
{
...
      ...{ // general case (internal patch)
            for(i=1; i<3; i++)
               for(j=1; j<3; j++)
                  for(k=0; k<3; k++)
                     patch[i][j][k]=ctrlpts[2*xstep+i][2*ystep+j][k];
            for(i=1; i<3; i++)
               for(k=0; k<3; k++) {
                  patch[i][0][k]=(ctrlpts[2*xstep+i][2*ystep][k]
                                  +ctrlpts[2*xstep+i][2*ystep+1][k])/2.0;
                  patch[i][3][k]=(ctrlpts[2*xstep+i][2*ystep+2][k]
                                  +ctrlpts[2*xstep+i][2*ystep+3][k])/2.0;
                  patch[0][i][k]=(ctrlpts[2*xstep][2*ystep+i][k]
                                  +ctrlpts[2*xstep+1][2*ystep+i][k])/2.0;
                  patch[3][i][k]=(ctrlpts[2*xstep+2][2*ystep+i][k]
                                  +ctrlpts[2*xstep+3][2*ystep+i][k])/2.0;
               }
            for(k=0; k<3; k++) {
               patch[0][0][k]=(ctrlpts[2*xstep][2*ystep][k]
                              +ctrlpts[2*xstep+1][2*ystep][k]
                              +ctrlpts[2*xstep][2*ystep+1][k]
                              +ctrlpts[2*xstep+1][2*ystep+1][k])/4.0;
               patch[3][0][k]=(ctrlpts[2*xstep+2][2*ystep][k]
                              +ctrlpts[2*xstep+3][2*ystep][k]
                              +ctrlpts[2*xstep+2][2*ystep+1][k]
                              +ctrlpts[2*xstep+3][2*ystep+1][k])/4.0;
               patch[0][3][k]=(ctrlpts[2*xstep][2*ystep+2][k]
                              +ctrlpts[2*xstep+1][2*ystep+2][k]
                              +ctrlpts[2*xstep][2*ystep+3][k]
                              +ctrlpts[2*xstep+1][2*ystep+3][k])/4.0;
               patch[3][3][k]=(ctrlpts[2*xstep+2][2*ystep+2][k]
                              +ctrlpts[2*xstep+3][2*ystep+2][k]
                              +ctrlpts[2*xstep+2][2*ystep+3][k]
                              +ctrlpts[2*xstep+3][2*ystep+3][k])/4.0;
            }
         }

...
}
```

A word to the wise...

This discussion has only covered cubic and bicubic splines, because these are readily provided by OpenGL evaluators. OpenGL also has the capability of providing NURBS (non-uniform rational B-splines) but these are beyond the scope of this discussion. Other applications may find it more appropriate to use other kinds of splines, and there are many kinds of spline curves and surfaces available; the interested reader is encouraged to look into this subject further.

Chapter 16: Per-Pixel Operations

Prerequisites

Experience with geometric operations in an algebraic setting.

Introduction

So far in the book we have focused on graphics APIs that are strongly polygon-oriented, and have learned how to use this kind of tools to make effective images. However, this is not the only kind of computer graphics that can be done, and it is also not the only kind of computer graphics that can be effective.

Another kind of computer graphics works with a geometric model to determine the color of each pixel in a scene independently. We call this *per-pixel* graphics, and there are several different approaches that can be taken to this. There is ray casting, a straightforward approach to generating an image in which rays are generated from the eye point through each pixel of a virtual screen, and the closest intersection of each ray with the geometry of the scene defines the color of that pixel. There is ray tracing, a more sophisticated approach that begins in the same way as ray casting, but you may generate more than one ray for each pixel and when the rays intersect with the geometry, they may define secondary rays by reflection or refraction that will help define the color that the pixel will eventually be given. There are also several uses of pixels as ways to record some computational value, such as one might find in a study of iterated functions or of fractals. With a wide variety of ways to take advantage of this kind of graphics, we believe it should be included in a beginning course.

Definitions

The terms *ray tracing* and *ray casting* are not always well-defined in computer graphics. We will define ray casting as the proces of generating an image from a model by sending out one ray per pixel and computing the color of that pixel by considering its intersection with a model with no shadow, reflection, or refraction computation. We will define ray tracing as an extension of ray casting without limitations: multiple rays may be cast for each pixel as needed, and may use any kind of shadow, refraction, and/or reflection computation as you wish. We will discuss both these processes, although we will not go into much depth on either and will refer you to a resource such as the Glassner text or a public-domain ray tracer such as POVRay to get more experience.

Ray casting

If we recall the standard viewing setup and viewing volume for the perspective transformation from Chapter 1, we see that the front of the viewing frustum can be an effective equivalent of the actual display screen. We can make this more concrete by applying a reverse window-to-viewport operation and, in effect, create a screen-to-viewplane operation that will give us a virtual screen at the front of the frustum.

To do this, we can take advantage of the idea of the perspective viewing volume that we saw in Chapter 1. We can create the viewing transformation as we illustrated in the mathematics discussion in Chapter 4, which allows us to consider the eye point at the origin and the front of the viewing volume in the plane $z = -1$. Just as the perspective projection in OpenGL is defined in terms of the field of view α and the aspect ratio ρ, we can determine the coordinates of the front of the view volume as $x = \tan(\alpha)$ and $y = \rho * x$. This space is the virtual screen, and you can divide up the screen into pixels by using a step size of $2*x/N$ where N is the number of pixels across the

horizontal screen. A virtual screen in world space, along with the eyepoint and a sample ray, is shown in Figure 15.1. In order to manange the aspect ratio you have set, you should also use the same step size for the vertical screen. The final parameters of the OpenGL perspective projection are the front and back clipping planes for the view; you may use these parameters to limit the range where you will check for intersections or you may choose to use your model with no limits.

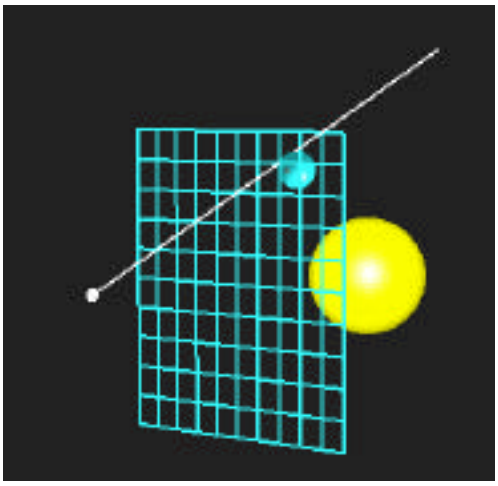


Figure 15.1: the eye point and virtual screen in 3D space

The key point of ray casting is to generate a ray, which you can model as a parametric line segment with only positive parameter values, from the eye point through each pixel in the virtual screen. You may choose what point you want to use in the pixel because each pixel actually corresponds to a small quad determined by adjacent horizontal and vertical lines in the virtual grid. In the general discussion it doesn't matter which point you choose, but you might start with the middle point in the quad. The ray you generate will start at the origin and will go through a point $(x,y,1)$, so the parametric equations for the ray are (xt, yt, t).

Once you have the parametric equation for a ray, you must determine the point in your model where the ray intersects some object. If there is no such point, then you simply get the background color; if there is an intersection, then you must determine the nearest intersection point. The pixel is then given the color of your model at that point of intersection. The ray-object intersection problem is the heart of the ray casting problem, and a great deal of time is spent in calculating intersections. We discussed collision testing in Chapter 4, and the ray-object intersection is just a special case of collision testing. The easiest intersection test is made with a sphere, where you can use the formula for the distance between a point and a line to see how close the line comes to the center of the sphere; if that distance is less than the radius of the sphere, there is an intersection and a simple quadratic equation will give you the intersection points.

Determining the color depends a great deal on the model you are viewing. If your model does not use a lighting model, then each object will have its own color, and the color of the pixel will simply be the color of the object the ray intersects. If you do have lighting, however, the local lighting model is not provided by the system; you must develop it yourself. This means that at the point where a ray meets an object, you must do the computations for ambient, diffuse, and specular lighting that we described in Chapter 9. This will require computing the four vectors involved in local lighting: the normal vector, eye vector, reflection vector and light vector. The eye vector is simple because this is the ray we are using; the normal vector is calculated using analytic techniques or geometric computations just as it was for the OpenGL `glNormal*(...)` function, the reflection vector is calculated as in Chapter 4, and it is straightforward to create a vector from

the point to each of the lights in the scene. Details of the lighting computations are sketched in Chapter 9, so we will not go into them in more detail here.

Because you are creating your own lighting, you have the opportunity to use more sophisticated shading models than are provided by OpenGL. These shading models generally use the standard ambient and diffuse light, but they can change the way specular highlights appear on an object by modifying the way light reflects from a surface. This is called *anisotropic shading* and involves creating a different reflection vector through a bidirectional reflection distribution function or BRDF. This kind of function takes the x- and y-components of the light vector and determines a direction that light will be reflected; the standard specular lighting computation is then done with that reflection direction vector. Details are much more complex than are appropriate here, but you can consult advanced references for more information.

Ray casting can also be used to view pre-computed models. Where each ray intersects the model, the color of that point is returned to the pixel in the image. This technique, or a more sophisticated ray-tracing equivalent, is often used to view a model that is lighted with a global lighting model such as radiosity or photon mapping. The lighting process is used to calculate the light at each surface in the model, and other visual properties such as texture mapping are applied, and then for each ray, the color of the intersection point is read from the model or is computed from information in the model. The resulting image carries all the sophistication of the model, even though it is created by a simple ray casting process.

Ray casting has aliasing problems because we sample with only one ray per pixel; the screen coordinates are quite coarse when compared to the real-valued fine detail of the real world. We saw this in our images created with OpenGL and noted that there are some anti-aliasing capabilities provided by that API. These aliasing problems are inherent in our definition of ray casting and cannot be addressed in original image, but you can do some post-processing of the image to smooth it out. This can lose detail but you must decide whether the smoothness is worth it. The fundamental principle is to generate the image in the frame buffer and then save the buffer as a color array. This array may then be processed by techniques such as filtering to smooth out any aliasing.

Ray tracing

As we define it, the difference between ray casting and ray tracing is that ray tracing includes any or all of the techniques of reflection, refraction, shadows, and multiple rays per pixel.

Let's begin by discussing how to handle shadows. When you calculate the lighting model and include a light in the model, you will create a vector from the intersection point to that light. However, that light only participates in the diffuse and specular lighting calculations if the light actually reaches the surface; otherwise the point is in shadow for that light. So you simply cast a ray from the point in the direction of the light and see if it intersects anything between the point and the light. If it does, then the point is not illuminated by that light so there is no diffuse or specular contribution from the light; the point is in shadow from that light. However, you should note that this requires that you generate a new ray from an arbitrary point in the scene in an arbitrary direction and will require an extra set of ray/object intersection computations.

Ray tracing also knows how to work with reflective and transmissive surfaces: surfaces for which light is reflected without spreading and surfaces for which light goes through the surface and continues on the other side. Reflection is relatively easy to deal with because we have already met the concept of the reflection vector: if the incoming vector is P and the normal vector is N, then the reflection vector R is computed by $P–2(N•P)N$. When light goes through a surface and continues on the other side, the process is called refraction. Here the ray continues in the same plane but its direction is affected by the difference between the speed of light in the old medium outside the

surface and the new medium inside the surface. The relative speeds of light in the media are not, in fact, used directly, but the index of refraction η of each material is. The key formula for the direction of refracted light is given by Snell's law: if Θ_1 and Θ_2 are the angles between the ray and the normal for the incoming ray and outgoing ray, respectively, then $\sin(\Theta_1)/\sin(\Theta_2)=\eta_1/\eta_2$. From this you can compute the vector for the outgoing light. The behavior of reflected and refracted light are illustrated in Figure 15.2.

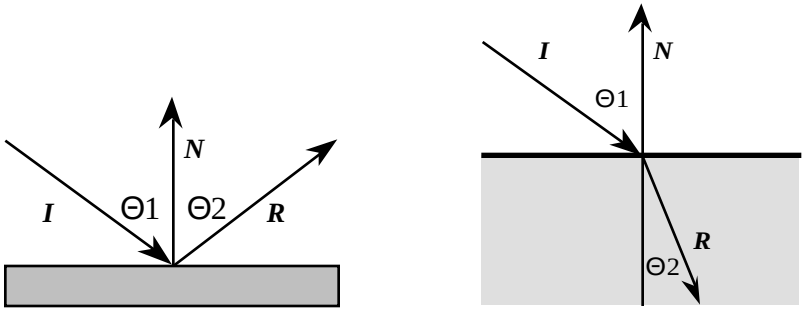


Figure 15.2: reflected (left) and refracted (right) rays where a ray intersects and object

Now once you have the vectors for reflected and refracted light, you must generate new rays from the point where the incoming ray meets the object, in the direction of the reflection vector and of the refraction vector. These are treated recursively just like the original ray, and eventually each will return a color value. These two values are combined with whatever color the object itself has, in the proportions by which the light was reflected, refracted, or simply returned diffusively.

The recursive generation of reflection and/or refraction rays are shown in the ray tree, as suggested in Figure 15.3, which includes reflected rays *R*, refracted rays *T*, and light rays *L*. You cannot, of course, go on forever allowing new reflection or refraction rays to be generated; you must define a mechanism to stop generating them eventually. This can be done by keeping track of the attenuation of light as only part of the light is reflected or refracted and stopping the recursion when a very small amount of light is actually being handled, or you can simply not allow any recursion past a certain level. But the recursive ray generation, with its need for a new set of ray/object intersection computations for rays that are generated from an arbitrary point, is certainly one of the issues that gives ray tracing the reputation for being quite slow.

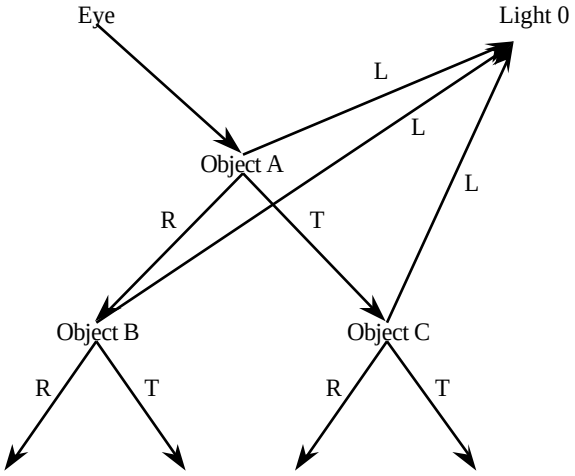


Figure 15.3: a ray tree

With the one ray per pixel of ray casting, we could not avoid aliasing problems and so above we briefly discussed a post-processing approach to reduce the effects of aliasing. This approach costs you some resolution in the final image, however, and we would like to avoid that. In ray tracing, you can generate multiple rays per pixel to do a better job or determining the color that best represents that pixel's contribution to the scene. You can generate these rays systematically, dividing the pixel into regular sub-pieces and generating a ray for each, or you can generate them stochastically, selecting random points in the pixel through which you generate rays. There are, of course, many variations on these themes, and you need to look at the literature to see more details. However, as you get the color value for each of these rays, you must reconstruct the color for the pixel from the colors of the individual rays. You may simply average these colors, or you may weight the colors across the pixel and compute the weighted average, probably with colors near the center having a larger weight than colors near the pixel edges.

Simple ray tracing programs are only moderately difficult to write, and in many technical computer graphics courses one of the student projects is to write a ray tracer. There are also several ray tracing programs that have been made available to the public. Of these, the POVRay program is probably the best known; it is available for most computer systems from http://www.povray.org. In fact, POVRay would also qualify as a graphics API because it has its own way to express geometry and control rendering, and you can write scene descriptions in its language that create excellent images. The source code for ray tracers is often available, and this is the case for POVRay in case you want to know all the details of this system. Figure 15.4 shows the output of the POV ray tracer on one of the benchmark scene definitions that comes with the system; note the many reflections and the shadow in the scene.

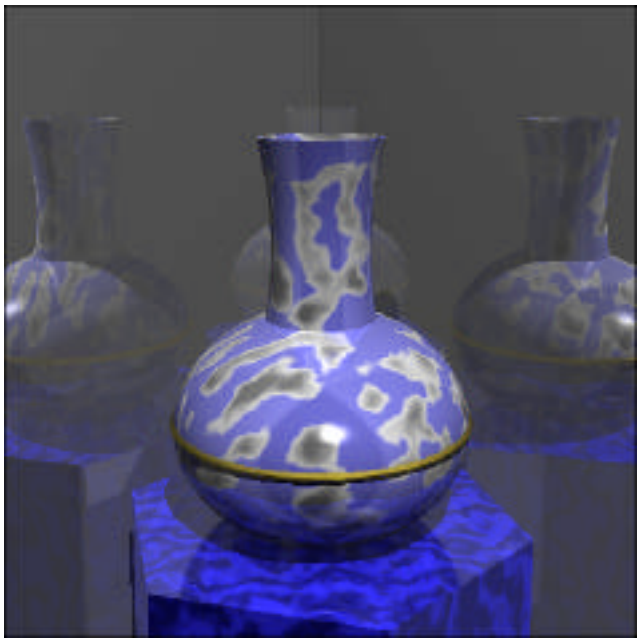


Figure 15.4: a ray-traced image created with POVRay

Ray tracing and global illumination

The concept of generating rays and identifying objects they intersect is not limited to ray casting and ray tracing. The global illumination technique of photon mapping uses a similar process to identify the illumination of each object in a scene. We will only sketch this very briefly, because

reconstructing the actual illumination of each object involves a number of sophisticated mathematical techniques that are beyond the scope of this discussion.

The basic process of photon mapping involves generating a large number of randomly-directed rays from each emissive light source in a scene. Each ray represents the path of a photon from the light source. When the ray meets an object, the object records the fact that a photon of a certain color has arrived, because the number of photons meeting an object is a measure of the illumination of that object. But because the main premise of global illumination is that every object not only receives illumination but also emits illumination, each photon that meets an object may also be emitted from the object, based on probability calculations that represent the material making up the object. If a photon is emitted, it is given a random direction and is given the color determined by the material. This process continues recursively for each photon until it ends because of probability computations or until a given number of steps has occurred.

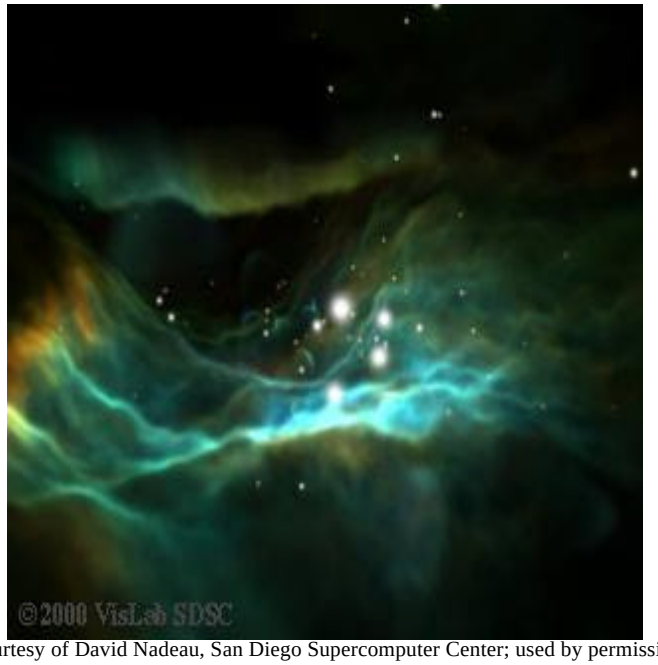
After the full set of photons is emitted and all the objects in the scene have accumulated information on the photon intersections, the scene is processed to determine an illumination value for each object. Like most global illumination techniques, the illumination only needs to be done once unless objects move in the scene. Once the illuminations are known, the scene can be rendered by ray casting or by other techniques to present it to the viewer. Because the illumination computation may not need to be re-done each time the image is rendered, it is possible to get good frame rates for displays such as walkthroughs in such scenes.

Volume rendering

An important application of the ideas of ray casting can be found in rendering volume data. As you will recall from Chapter 7, a real-valued function of three variables may be thought of as a volume with every point in 3D space being associated with a real value. This function may be analytic, or defined by a formula or process, or it may be data-derived, or defined by some experimental or constructed work.

In Chapter 7 we discussed some simple techniques for examining a real-valued function on a volume to display an isosurface (the surface where the function has a given value), but a more satisfactory approach may be to cast a set of rays into the volume and compute the points where each intersects the given isosurface. With each intersection, you determine the voxel in which the intersection occurs and a normal is computed by examining how the level surface meets the boundary of the voxel, and the usual lighting model can be applied. The resulting image is a much better presentation of a level surface in the volume.

We can use a ray-casting technique to do more than this simple isosurface study, however. Instead of dealing with a simple real function in the volume and building an image of an isosurface, you can create a much more complex model in which there is a rich environment in the volume, and you can apply whatever techniques are appropriate for the model along each ray to create an image of the volume. For example, a model of the Orion nebula was built based on data from the Hubble space telescope. At each point in the volume, the model defined both color and opacity based on the nature of the space within the nebula. As each ray is cast into this space it accumulates color from the objects it meets but also includes color from beyond the intersection, in an approach not unlike the blending operations we have seen. As objects are met by the ray, processed from front to back, the ray develops its color as (object color + (1-opacity)*(color from remaining ray)) until the accumulated opacity is essentially 1. The result of this work for the Orion model is seen in the wonderful image of Figure 15.5.



courtesy of David Nadeau, San Diego Supercomputer Center; used by permission

Figure 15.5: a volume visualization of a model of the Orion nebula

Fractal images

There are a number of kinds of processes that go under the name of fractal images. Here we will focus on two that are distinctive because they are usually presented with images where each pixel represents the behavior of the process at a given point. Other kinds of fractal images, such as artificial landscapes, are usually presented through polygon-based images and are handled with techniques that belong in the rest of the book. We mention the relation between the pixel-based images and the polygon-based images briefly later in the chapter.

There are many interesting issues in dynamical systems that can be explored with computer graphics. We mentioned them briefly in Chapter 7 and will look at them again here, but we will not go into any detail; see the references by Mandelbrot and Pietgen, as well as the general popular literature on fractals, for more information.

The key behavior we are considering is whether a sequence of complex quadratic functions $\{f_k(z)\}$ will converge or diverge as n increases when it is iterated as $f_{n+1}(z) = f_n(z)^2+c$. and $f_0(z) = z^2+c$. The functions will converge if there is a bound on $|f_k(z)|$ as k increases, while they will diverge if there is no such bound. We remind you that complex numbers are given by pairs of real numbers, (a, b), and are usually written $z = a + bi$, where $i^2 = -1$. So our arithmetic is defined by

$$(a + bi)^2 = (a^2-b^2)+2abi$$
$$(a + bi) + (c + di) = (a+c)+(b+d)i$$
$$|a + bi| = \text{sqrt}(a^2+b^2)$$

Now if we take the sequence $\{f_k(z)\}$ for different values of c, always starting with the initial value $z=0$, we are investigating the behavior of the parameter space $\{c\}$. The Mandelbrot set is the set of complex numbers c so that this sequence converges with this initial value. If $|f_k(z)| > 2$ for any value of k, then the sequence will diverge, so we simply ask ourselves whether $|f_k(z)| < 2$ for all

values of *k* up to a fairly large value such as, say, 500. If there is a value of *k* for which $|f_k(z)| > 2$, then the first such *k* is returned for the complex number *c*; if not, then the value of 0 is returned, and the Mandelbrot set is approximated by those complex numbers that have the value of 0.

To display this situation graphically, we identify a complex number (*a*+*b*i) with the 2D point (*a*, *b*), and we color the point with some color ramp according to the value we record as above for the complex number. This identification can also go the other way: with each point we can identify a complex number and apply the process above to determine a color. We can then take a 2D domain and create a grid on the domain that matches the dimensions of our window, as we describe below, apply the process above for each grid point, and color the corresponding pixel as above. An image of the Mandelbrot set, which is probably familiar to you because it has captured the imagination of many people, is shown in Figure 15.6, along with a detail showing some of the fantastic behavior of the convergence process when we examine a very small region. The full Mandelbrot set is shown for the complex numbers (*re* + *im* i) for *re* in [-1.5, 0.5] and *im* in [-1, 1], while the detail is the region with *re* in [.30, .32] and *im* in [.48, .50].

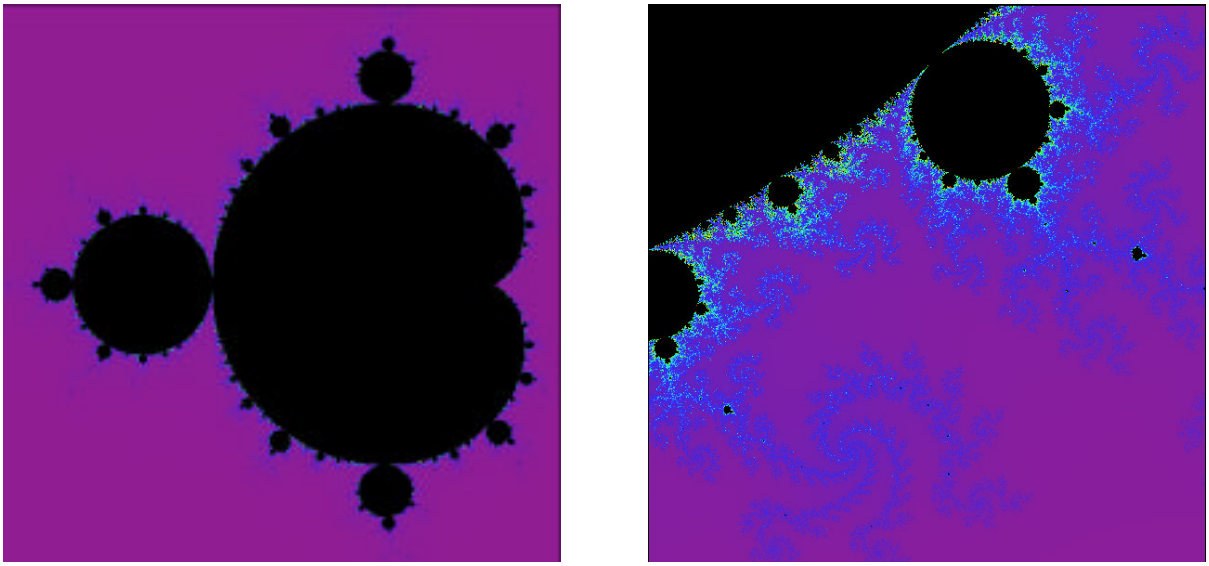


Figure 15.6: the full Mandelbrot set (left) and a detail (right)

We can ask a very similar question about the sequence $\{f_k(z)\}$ if we change the rules: choose a fixed value of *c*, compute the sequence for different values of *z*, and use the same coloring technique as above. The set of complex numbers *z* for which the sequence converges is called a *Julia set*. Julia sets are related to the Mandelbrot set because if you create a Julia set with any complex number in the Mandelbrot set, the Julia set is connected; if you do so with a complex number outside the Mandelbrot set, the Julia set is completely disconnected. Complex numbers inside the Mandelbrot set but very near the edge create very interesting and unusual Julia sets. Figure 15.7 shown the particular Julia set computed for the fixed point (–.74543, ,11301), and as we did with the Mandelbrot set, you can choose to display only a small part of the set and get even more detailed and fascinating images. It would be interesting to present a Mandelbrot set and allow a user to select a point in that space and then generate the corresponding Julia set.

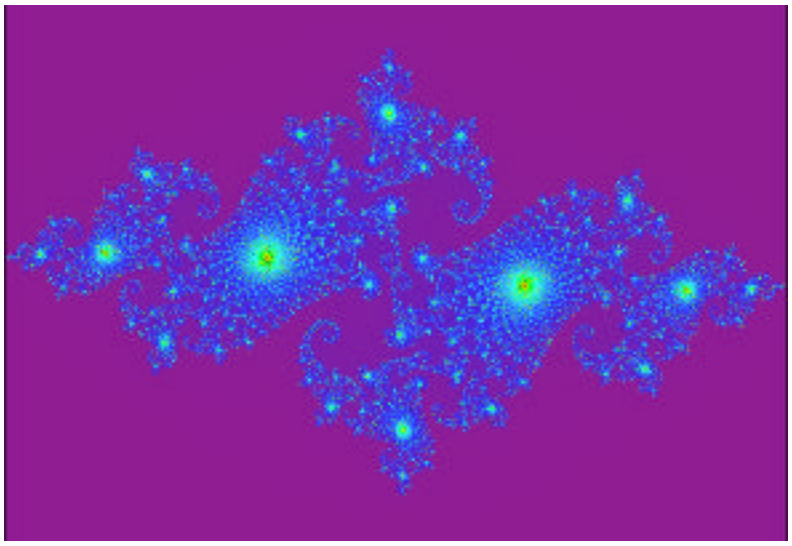


Figure 15.7: a Julia set for a particular fixed value *c*

Iterated function systems

The concept of an iterated function system (IFS) includes a number of kinds of operations, but we will focus our discussion on two of these. One is the contraction mapping operation. A contraction mapping is a function from a closed, bounded region to a closed bounded region with smaller area. Any contraction mapping has the property that if it is applied often enough, the original region will be mapped into a region of arbitrarily small area. Now if we start with a point $q=q_0$ and a set of contraction mappings $\{f_i\}$, and define $q_i=f_i(q_{i-1})$ where each function f_i is applied with probability p_i, then there will be a well-defined set of points called the *attractor* of the IFS so that for sufficiently large values of i, the point q_i will be arbitrarily close to a point in the attractor. We have already seen one such example in the 3D Sierpinski attractor we discussed in Chapter 7, where the four linear contraction functions, written in vector form, are $f_i(p) = (p+p_i)/2$, for $\{p_i\}$ the four vertices of a tetrahedron. Each function is applied to a point p with probability 1/4 and, after a relatively modest number of iterations, the resulting point is an excellent approximator of the Sierpinski tetrahedron. Because we have already seen the results of this IFS, we will not discuss this example further.

Another contraction mapping IFS that could be of interest is the 2D mapping that models a fern leaf. Because this is a 2D mapping that involves presenting the points of attraction of the IFS, we can visualize its effects using the per-pixel operations in this chapter. The mapping for the fern is defined by the linear functions below, with the associated probability for each:

$f_0(x,y) = (0, .16y)$ $p = .01$
$f_1(x,y) = (.85x+.04y, -.04x+.85y+1.6)$ $p = .85$
$f_2(x,y) = (.20x–.26y, .23x+.22y+1.6)$ $p = .07$
$f_3(x,y) = (-.15x+.28y, .26x+.24y+.44)$ $p = .07$

The result, after several iterations, is seen in Figure 15.8.

Figure 15.8: the fren image generated by the iterated function system above

Another kind of IFS is created by defining a geometric structure recursively, replacing a simple entity by a more complex entity with the same beginning and end vertices. The 2D version of the blancmange function of Chapter 7 may be defined in this way by starting with a single line segment and replacing each segment by a pair of lines as shown in Figure 15.9. The "everywhere continuous" property of the resulting limit function is because the limit function is the limit of a converging sequence of uniformly continuous functions; the "nowhere differentiable" property is because between if you choose any pair of values, no matter how close, there will be a line segment between them for some iteration of this process and the next iteration will have a sharp angle in the interval.

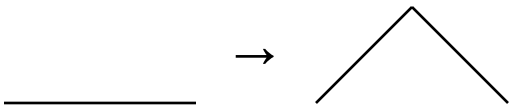


Figure 15.9: the generating process for the 2D blancmange function

Of course, this kind of function is not presented by the simple per-pixel operations of this chapter but by the kind of geometric operations we have discussed througout this book. As another example, let us consider the "dragon curve" that is defined by replacing a simple line segment by two segments, as was the case with the blancmange curve, but differing from that curve by putting the two segments offset to alternating sides of the line. This generating operation is shown in Figure 15.10.

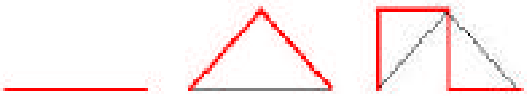


Figure 15.10: the dragon curve generator

The resulting curve continues with as many iterations as you want, but when stopped at the 10^{th} iteration the single dragon curve is shown as the left-hand image in Figure 15.11. A fascinating property of the dragon curve is the way these curves fill the space around a given point. This is shown in the right-hand image of the figure, where four curves of different colors meet at a point.

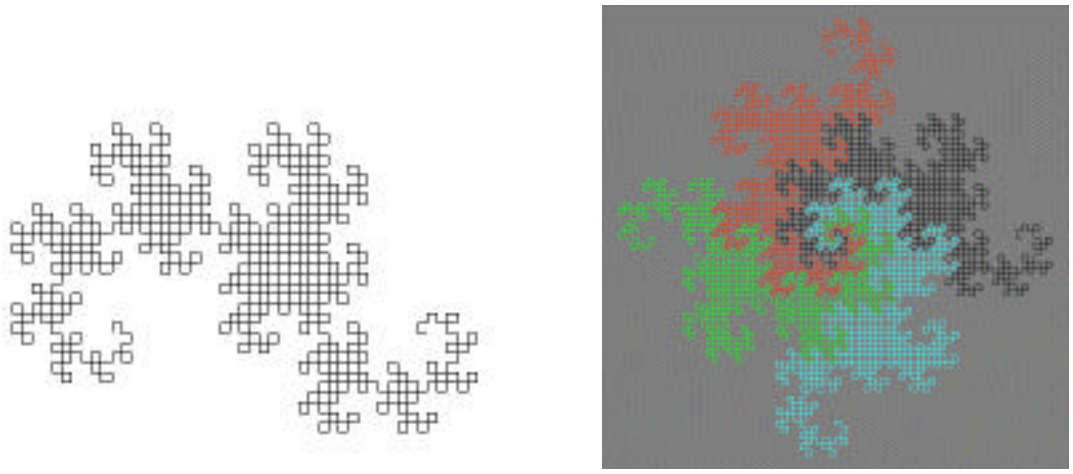


Figure 15.11: a simple dragon curve, and the set of four curves around a fixed point

Per-pixel operations supported by OpenGL

There are two ways you could use OpenGL to display an image that is the result of per-pixel operations. The first is to use the GL_POINTS rendering mode, with a color determined for each point. This would involve a looping operation something like the following:

```
glBegin(GL_POINTS)
   for row = 0 to N-1
      for column = 0 to M-1
         calculate point (x,y) for pixel (M,N)
         calculate color for point (x,y); return color
         glColor(color)
         glVertex2i(x,y)
glEnd()
```

This operation implies that we define a window that is M pixels wide and N pixels high and that we set a 2D orthogonal projection with dimensions that match the window, but that is straightforward: in `main()` we have `glutInitWindowSize(M,N)` and in `init()` we have `gluOrtho2D(0.,(float)M,0.,(float)N)`. This operation does not support any window reshapes that change the size of the window because it is totally tied to the original window dimensions, unless your calculation of the virtual screen coordinate is flexible and can use new dimensions. The images in Figures X.5, X.6, and X.7 were created with this kind of operation, and the actual detailed code for this inner computation in the Mandelbrot set is given below.

```
xstep = (XMAX - XMIN)/(float)(WS-1); //WS=window size in pixels
ystep = (YMAX - YMIN)/(float)(WS-1);
glBegin(GL_POINTS);
for (i = 0; i < WS; i++) {
   x = XMIN + (float)i * xstep;
   for (j = 0; j < WS; j++) {
      y = YMIN + (float)j * ystep;
      test = testConvergence(x,y);
      // ITERMAX = maximum no. of iterations
      // colorRamp function behaves as color ramps in Chapter 6
      colorRamp((float)test/(float)ITERMAX);
      glColor3fv(myColor);
      glVertex2f((float)i,(float)j);
   }
}
glEnd();
```

There are a couple of other approaches to displaying the results of this kind of calculation. One is to create a domain of an appropriate size to present the detail you want and set a color value for each point in the domain as you did in the per-pixel operations. You can then graph the function on the domain as a set of 2D polygons instead of just using pixels. If you use smooth shading, you can get an image that shows the behavior more smoothly than per-pixel operations would, although you must be careful not to disguise essential discontinuities with the smooth shading. This would probably not be a good approach to presenting Mandelbrot and Julia sets, for example. While this is not a per-pixel operation, it is essentially the same approach we used for these operations but takes advantage of the extra capabilities of a graphics API.

Another way to present your results would be to define a height each point in the domain using the same kind of technique that you used to define a color, and create a 3D surface with these heights. This is also not a per-pixel operation, of course, but it opens many opportunities for creating interesting presentations of fractals and similar things as shown in Figure 15.12. Here we have taken the value k of the complex dynamic systems problem and have computed a height as 1 if the value of k is 0, or $1–1/k$ if k is non-zero. This figure shows the Mandelbrot set from Figure 15.6 as a plateau whose sides drop off as the dynamic system diverges. The stepwise nature of the sides comes from the fact that the number of iterations to divergence is an integer and so the height of the surface is discontinuous.

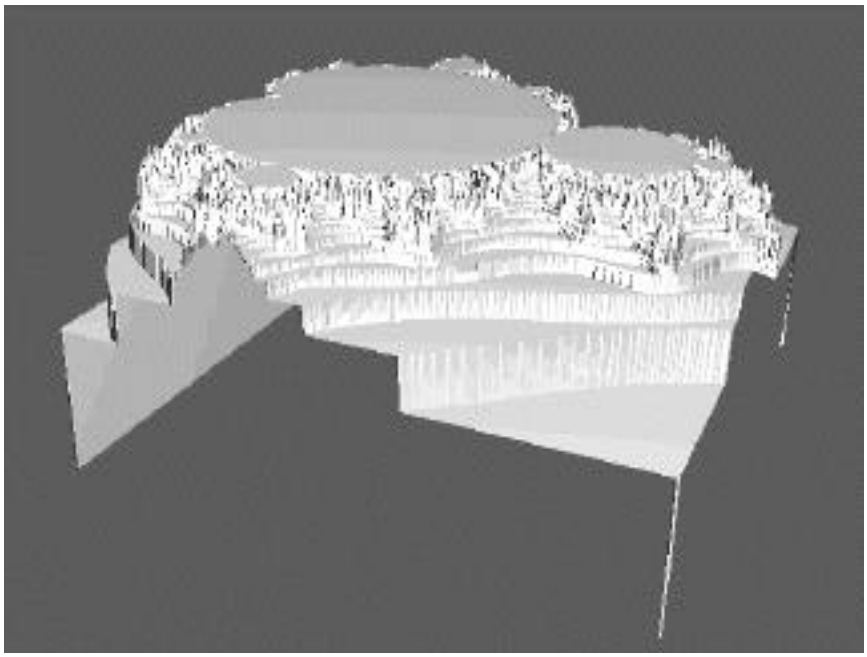


Figure 15.9; a surface presentation of the Mandelbrot set and its surrounding region

Chapter 17: Hardcopy

Prerequisites

An understanding of the nature of color and visual communication, and an appreciation of what makes an effective image.

Introduction

You have worked hard to analyze a problem and have developed really good models and great images or animations that communicate your solution to that problem, but those images and animations only run on your computer and are presented on your screen. Now you need to take that work and present it to a larger audience, and you don't want to lose the control over the quality of your work when you take it to a medium beyond the screen. In this chapter we talk about the issues you will face when you do this.

Definitions

Computer graphics hardcopy is output in a fixed medium that can be taken away from the context in which it was created, and can be communicated to your audience without that original context. There are several ways this can be done, but the basic idea is that any kind of medium that can carry an image is a candidate for hardcopy. These can be physical (paper, sculptural) or digital (images, video). Each of these media has its own issues in terms of its capability and how you must prepare your images for the medium. In this chapter we will discuss some of the more common hardcopy media and give you an idea of what you must do to use each effectively.

Creating hardcopy can mean creating a digital record of the work that can be sent to some sort of output device. That device may be actually attached to the computer, such as a printer or a film recorder, or it may be a device to which we communicate data by network, disk, or CD-ROM. So part of the discussion of graphics hardcopy will include a description of the way data must be organized in order to communicate with external production processes.

Digital images:

If your work creates single images, then one way to communicate these images is by saving them to a file that is organized by some generally-readable graphics file format. We have seen in earlier chapters that we can save the contents of our color buffer into an internal color array, and this array can be processed to create an image file in any of the standard formats—GIF, TIFF, JPEG, PNG, or others. If you simply write the array to a file of unsigned characters, you will create what is sometimes called a *raw RGB* file, and there are generally-available libraries of tools for format conversion that can translate such files into the more generally useable formats above. If you do not have a library of file format conversions around, then you can open a raw RGB file with Photoshop™ and save it with any of the standard formats; in essence you are using Photoshop as a file format conversion utility.

If you are not familiar with these file formats, perhaps a quick description is in order. *GIF* stands for Graphics Image Format, and it stores images in 8-bit indexed color with some lossless compression. However, the GIF file format involves Lempel-Ziv-Welch file compression, and this is a patented process so you must license it if you create any software that implements this algorithm. *TIFF* is the Tagged Image File Format, a very general format that stores images with whatever color depth you need. Generally there is little file compression with TIFF files and they can be very large. *JPEG* is a file format that often uses a lossy compression based on the discrete cosine transformation. JPEG is strong on natural images but weak on images that have lines or

sharp edges. Its name is based on its developers: the Joint Photographic Experts Group. *PNG* stands for Portable Network Graphics, a file format that was created in an effort to replace GIF as a widely-used format for images on the networks. One of the main aspects of PNG is that it does not use any legally encumbered data compression. It includes the capabilities of GIF but adds full-color support, including alpha values, as well as 16-bit grayscale support.

Print:

One version of printed hardcopy is created by a standard color printer that you can use with your computer system. Because these printers put color on paper, they are usually CMYK devices, as we talked about in the color chapter, but the printer driver will usually handle the conversion from RGB to CMYK for you. In order of increasing print quality, the technologies for color output are

- inkjet, where small dots of colored ink are shot onto paper and you have to deal with dot spread and over-wetting paper as the ink is absorbed into the paper,
- wax transfer, where wax sticks of the appropriate colors are melted and a thin film of wax is put onto the paper, and
- dye sublimation, where sheets of dye-saturated material are used to transfer dyes to the paper.

These devices have various levels of resolution, but in general each has resolution somewhat less than a computer screen. All these technologies can also be used to produce overhead foils for those times when you have only an overhead projector to present your work to your audience.

Print can also mean producing documents by standard printing presses. This kind of print has some remarkably complex issues in reproducing color images. Because print is a transmissive or subtractive medium, you must convert your original RGB work to CMYK color before beginning to develop printed materials. You will also need to work with printing processes, so someone must make plates of your work for the press, and this involves creating separations as shown in Figure 16.1 (which was also shown in the chapter on color). Plate separations are created by

Figure 16.1: separations for color printing

masking the individual C, M, Y, and K color rasters with a screen that is laid across the image at a different angle for each color; the resulting print allows each of the color inks to lay on the paper with minimal interference with the other colors. A screen is shown, greatly enlarged, in Figure

16.2, where the enlargement is so great that you can see the angles of the screens for the C, M, Y, and K components. (You should look at a color image in print to see the tell-tale rosettes of standard separations.) There are other separation technologies, called stochastic separations, that dither individual dots of ink to provide more colored ink on the page and sharper images without interference, but these have not caught on with much of the printing world. Creating separations for color-critical images is something of an art form, and it is strongly suggested that you insist on high-quality color proofs of your work. You must also plan for a lower resolution in print than in your original image because the technologies of platemaking and presses do not allow presses to provide a very high resolution on paper.

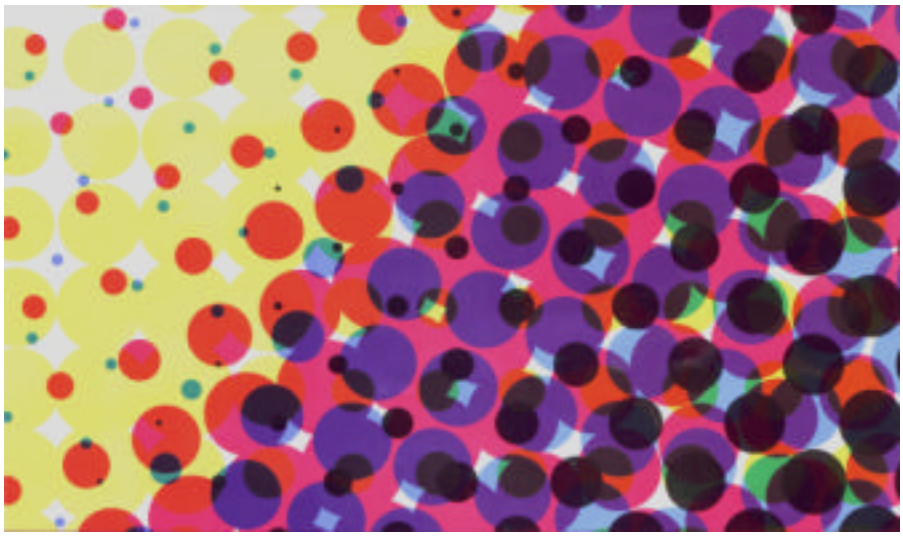


Figure 16.2: C, M, Y, and K screens in a color image, greatly enlarged

Film:

Sometimes you want to present the highest-quality images you can to an audience: the most saturated colors and the highest resolution. Sometimes you want to be sure you can present your work without relying on computer projection technology. In both cases, you want to consider standard photographic images from digital film recorders. These are devices that generate images using a very high-quality grayscale monitor, a color wheel, and a camera body and that work with whatever kind of film you want (usually slide film: Kodachrome, Ektachrome, or the like).

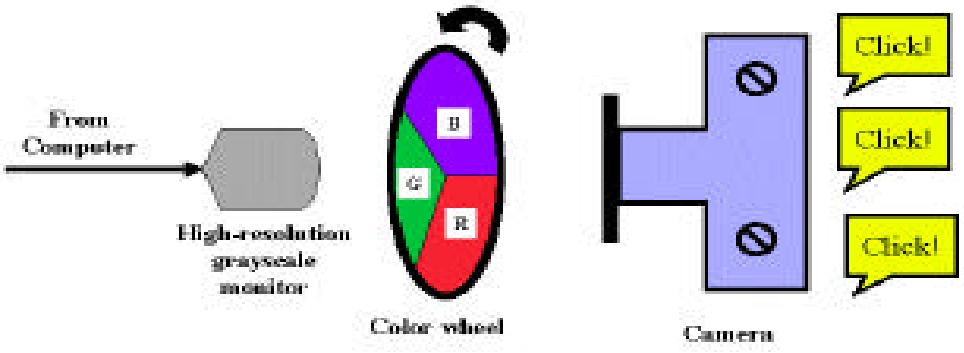


Figure 16.3: schematic of digital film recorder

A film recorder is organized as shown in Figure 16.3. The grayscale monitor generates the images for each color separately, and that image is photographed through a color wheel that provides the color for the image. Because a grayscale monitor does not need to have a shadow mask to separate the phosphors for different colors, and because the monitor can be designed to have a long neck and small screen to allow for extremely tight control of the electron beam, it can have extraordinary resolution; 8K line resolution is pretty standard and you can get film recorders with up to 32K lines. This allows you to generate your image at resolutions that would be impossible on the screen.

Film is much less of a problem than print, because you can work directly with the image and do not need to deal with separations, and you work with the usual RGB color model. Recall that slides produce their image by having light projected through them, so they behave as if they were an emissive medium like the screen. Your only issue is to deal with the resolution of the camera or to accept the interpolations the film recorder will use if you don't provide enough resolution.

Video:

video is a very important medium for your work, because it is the only medium available to show the motion that is so important to communicate many of your ideas. At the same time, it can be one of the most limited media available to you — at least until video leaves the first half of the 20th century and really comes into the 21st century. We will focus on NTSC video here, but there are similar issues for PAL or SECAM video, and if you are reading this in one of the areas where PAL or SECAM are the standards, you should check to see how much the comments here apply to you.

There are some important issues in dealing with video. The first is resolution: the resolution of NTSC video is much lower than even a minimal computer resolution. NTSC standards call for 525 interlaced horizontal scan lines, of which 480 are visible, so your planned resolution should be about 640 by 480. However, many television sets have adjustment issues so you should not ever work right against the edge of this space. The interlaced scan means that only half of the horizontal lines will be displayed every 1/30 second, so you should avoid using single-pixel horizontal elements to avoid flicker; many television sets have poorly-converged color, so you should also avoid using single-pixel vertical elements to they will not bleed into each other. In fact, you will have the best results for video if you design your work assuming that you have only half the resolution noted above.

A second issue in video is the color gamut. Instead of being composed of RGB components, the NTSC television standard is made up of significant compromises to acount for limited broadcasting bandwidth and the need to be compatible with black-and-white television (the NTSC standard dates from the late 1930s, before the wide-spread advent of color television or the advent of modern electronics and other technology). The NTSC color standard is a three-component model called the YIQ standard, but the three components are entirely focused on video issues. The Y component is the luminance (or brightness), and it gets most of the bandwidth of the signal. The I component is an orange-to-blue component, and it gets a little more than 1/3 of bandwidth of the Y component. The Q component is a purple-to-green component, and it gets a little more than 1/3 of the I component. The best color you can get in video always seems to be under-saturated, because that is part of the compromise of dealing with the technology available. To be more precise, the following table shows the bandwidth and the horizontal resolution for each of the components of the video image:

Component	Bandwidth	Resolution/scanline
Y	4.0 Mhz	267
I	1.5 Mhz	96
Q	0.6 Mhz	35

In order to get the best possible horizontal resolution from your image, then, you need to be sure that the elements that vary across the line have differing luminance, and you should focus more on the orange-to-blue component than on the purple-to-green component. If you want to understand how your colors vary in YIQ, the following conversion matrix should help you evaluate your image for video:

$$\begin{vmatrix} Y \\ I \\ Q \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{vmatrix} \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

The question of video is complicated by the various digital video formats, such as QuickTime and MPEG, that require computer mediation to be played back. Digital video is RGB, so it does not have many of the problems of NTSC until it is actually played on a television screen, and there are television sets that will handle increasingly-high quality video. In fact, MPEG II is the video standard for DVD, and there are self-contained DVD players, so this provides one alternative to doing your own conversion to NTSC.

In the longer term, television will be moving to native digital formats and the HDTV standards will support direct RGB color and higher-resolution, non-interlaced images, so we look forward to this discussion becoming antiquated. For the time being, however, you may need to put up with creating images that will make your colleagues ask, “That looks terrible! Why are you doing that?” Figure 16.4 is a photograph of a video image that shows the problems with color and resolution. If they understand that you’re going to video, however, they’ll understand.

Figure 16.4: screen capture of an NTSC video image.
Note the relatively low resolution and limited colors.

Digital video:

Creating a digital video from an animation is straightforward with the right tools. If you are creating an animation, for example, you may generate a single frame of the animation in a window on the screen and then use a function such as we described in earlier chapters to save the contents of the window to an array, which can then be written to an image file, possibly with a name that represents the frame number of that image in the overall animation. After you have completed

running your set of animation segments, and have created the set of individual frames, you may import them into any of a number of tools that will allow you to save them as a digital movie in QuickTime or MPEG format. There are several levels of MPEG format, with each level having a higher compression than its predecessor. Many of these tools will also allow you to add a sound track, do transitions from one animation sequence to another, or add subtitles or other text information to the movie. And when you finish developing the video, you can write it to a CD-R or even DVD-R disk to share with others.

3D object prototyping:

There are times when having an image of an object simply isn't enough, when you need to be able to run your fingers over the object to understand its shape, when you need to hold two objects together to see how they fit, or when you need to see how something is shaped so you can see how it could be manufactured. This kind of 3D object prototyping is sometimes called "3D printing" and is done by special tools. You can view the resulting object as a prototype of a later manufactured object, or you can view it as a solid representation of your graphic image. Figure 16.5 shows photographs of the (3,4)-torus as created by several of these 3D printing techniques, as noted in the figure caption. The contact information for each of the companies whose products were used for these hardcopies is given at the end of the chapter. There are, of course, other older technologies for 3D hardcopy that involve creating a tool path for a cutting tool in a numerical milling machine and similar techniques, but these go beyond the prototyping level.

There are several kinds of technologies for creating these prototype objects, but most work by building up a solid model in layers, with each layer controlled by a computation of the boundary of the solid at each horizontal cutting plane. These boundaries are computed from information on the faces that bound the object as represented in information presented to the production device. The current technologies for doing such production include the following:

- The Helisys LOM (Laminated Object Manufacturing) system lays down single sheets of adhesive-backed paper and cuts the outline of each layer with a laser. The portion of the sheets that is outside the object is scored so that the scrap to be removed (carefully!) with simple tools, and the final object is lacquered to make it stronger. It is not possible to build objects that have thin openings to the outside because the scrap cannot be removed from the

Figure 16.5a: the torus created by the LOM system

internal volumes. LOM objects are vulnerable to damage on edges that are at the very top or bottom of the layers, but in general they are quite sturdy. Figure 16.5a shows the torus created with the LOM system; note the rectangular grid on the surface made by the edges of the scrap scoring, the moiré pattern formed by the burned edges of the individual layers of paper in the object, and the shiny surface made when the object is lacquered.

- The Z-Corp Z-402 system lays down a thin layer of starch powder and puts a liquid binder (in the most recent release, the binder can have several different colors) on the part of the powder that is to be retained by the layer. The resulting object is quite fragile but is treated with a penetrating liquid such as liquid wax or a SuperGlue to stabilize it. Objects built with a wax treatment are somewhat fragile, but objects built with SuperGlue are very strong. Because the parts of the original object that are not treated with binder are a simple powder, it is possible to create objects with small openings and internal voids with this technology. Figure 16.5b shows the torus created with the LOM system; note the very matte surface that is created by the basic powder composition of the object.

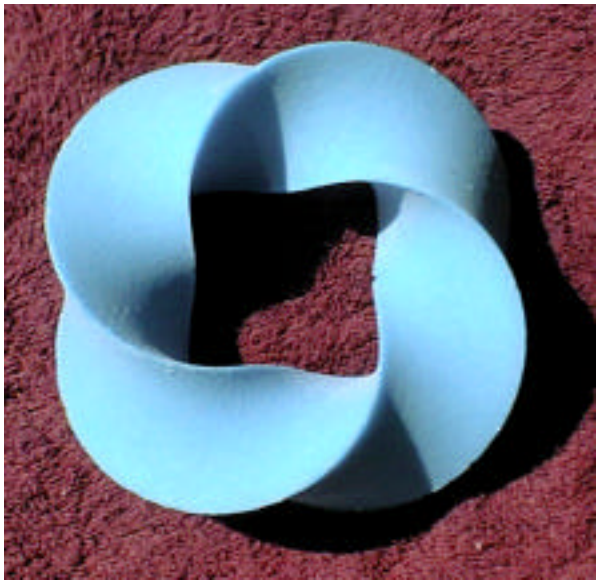


Figure 16.5b: the torus created by the Z-Corp system

- The 3D Systems ThermaJet system builds a part by injecting a layer of liquid wax for each layer of the object. Such parts must include a support structure for any regions that overhang the object's base or another part of the object, and this support can either be designed when the object is designed or provided automatically by the ThermaJet system. Because the object is made of wax it is not stable in heat, is not very strong, and is subject to breakage on any sharp edges. The need for a support structure makes it difficult to include voids with small openings to the outside. Also because of the support structure, the bottom part of an object needs to be finished by removing the structure and smoothing the surface from which this was removed. Figure 16.5c shows the torus as created by the ThermaJet system; note the slightly shiny surface of the wax in the object. The color depends on the color of the available wax; other colors besides gray are possible.

Figure 16.5c: the torus created by the 3D Systems ThermaJet system

- The 3D Systems stereolithography system creates an object by building up thin layers of a polymer liquid and hardening the part of that layer that is to be retained by scanning it with a laser beam. As with the ThermaJet system, this requires a very solid support structure for parts of the object, particularly because there is a small contraction of the polymer material when it is treated with the laser. The support structure must be removed from the object after it is completed, so some finishing work is needed to get fully-developed surfaces. The polymer liquid can readily be drained from any interior spaces if there is an opening to the outside, so this technology handles non-convex objects well. The polymer is very strong after it is hardened after the shaping is complete, so objects created with this technology are very sturdy. Figure 16.5d shows the torus as created by the stereolithography system.

Figure 16.5d: the torus created by the 3D Systems stereolithography system

The STL file

One thing all these 3d prototyping technologies have in common is that they all take data files in the STL (stereolithographic) file format in order to control their operations. This is a very simple file format that is easy to generate from your graphics program. The STL file for the (3.4)-torus is 2,459,957 bytes long and the first and last portions of the file are shown below. The file is organized by facets, and with each facet you have an optional normal and a list of the vertices of the facet; if you create your model in a way that will let you generate the explicit coordinates of your vertices, you can simply write the contents of the STL file instead of calling the graphics output functions. The only critical detail is that vertices on two triangles that are expected to align, such as vertices for boundary triangles, must have exactly the same values; most 3D prototyping systems are extremely fussy about small gaps. You should probably retain boundary values to be used for these vertices instead of relying on computations that can have very slight roundoff errors. The details of the STL file format are included as an appendix to these notes.

```
solid
  facet normal -0.055466 0.024069 0.000000
    outer loop
      vertex -5.000010 -0.000013 -1.732045
      vertex -5.069491 -0.160129 -1.688424
      vertex -5.000009 -0.000013 -1.385635
    endloop
  endfacet
  facet normal -0.055277 0.019635 0.002301
    outer loop
      vertex -5.069491 -0.160129 -1.688424
      vertex -5.000009 -0.000013 -1.385635
      vertex -5.054917 -0.159669 -1.342321
    endloop
  endfacet
  ...
  facet normal -0.055466 -0.024069 0.000000
    outer loop
      vertex -5.000009 0.000014 1.385635
      vertex -5.069491 0.160130 1.688424
      vertex -5.000010 0.000014 1.732045
    endloop
  endfacet
endsolid
```

A word to the wise...

The quick summary of this chapter is to know what your eventual medium will be, and to design for that medium when you plan your image or visualization. And be prepared to experiment as you work on your design, because some of these hardcopy media simply take experience that no notes or text can ever give you.

Appendices

This section contains the details of some file formats that have been used in examples in these notes. They are included for the student who wants to work on projects that use such file formats.

Appendix I: PDB file format

The national Protein Data Bank (PDB) file format is extremely complex and contains much more information than we can ever hope to use for student projects. We will extract the information we need for simple molecular display from the reference document on this file format to present here. From the chemistry point of view, the student might be encouraged to look at the longer file description to see how much information is recorded in creating a full record of a molecule.

There are two kinds of records in a PDB file that are critical to us: atom location records and bond description records. These specify the atoms in the molecule and the bonds between these atoms. By reading these records we can fill in the information in the internal data structures that hold the information needed to generate the display. The information given here on the atom location (ATOM) and bond description (CONECT) records is from the reference. There is another kind of record that describes atoms, with the keyword HETATM, but we leave this description to the full PDB format manual in the references.

ATOM records: The ATOM records present the atomic coordinates for standard residues, in angstroms. They also present the occupancy and temperature factor for each atom. The element symbol is always present on each ATOM record.

Record Format:

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	"ATOM "	
7 - 11	Integer	serial	Atom serial number.
13 - 16	Atom	name	Atom name.
17	Character	altLoc	Alternate location indicator.
18 - 20	Residue name	resName	Residue name.
22	Character	chainID	Chain identifier.
23 - 26	Integer	resSeq	Residue sequence number.
27	AChar	iCode	Code for insertion of residues.
31 - 38	Real(8.3)	x	Orthogonal coordinates for X in Angstroms.
39 - 46	Real(8.3)	y	Orthogonal coordinates for Y in Angstroms.
47 - 54	Real(8.3)	z	Orthogonal coordinates for Z in Angstroms.
55 - 60	Real(6.2)	occupancy	Occupancy.
61 - 66	Real(6.2)	tempFactor	Temperature factor.
73 - 76	LString(4)	segID	Segment identifier, left-justified.
77 - 78	LString(2)	element	Element symbol, right-justified.
79 - 80	LString(2)	charge	Charge on the atom.

The "Atom name" field can be complex, because there are other ways to give names than the standard atomic names. In the PDB file examples provided with this set of projects, we have been careful to avoid names that differ from the standard names in the periodic table, but that means that we have not been able to use all the PDB files from, say, the chemical data bank. If your chemistry program wants you to use a particular molecule as an example, but that example's data file uses other formats for atom names in its file, you will need to modify the `readPDBfile()` function of these examples.

Example:

```
         1       2       3       4       5       6       7       8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
ATOM      1  C           1      -2.053   2.955   3.329  1.00  0.00
ATOM      2  C           1      -1.206   3.293   2.266  1.00  0.00
ATOM      3  C           1      -0.945   2.371   1.249  1.00  0.00
ATOM      4  C           1      -1.540   1.127   1.395  1.00  0.00
ATOM      5  C           1      -2.680   1.705   3.426  1.00  0.00
ATOM      6  C           1      -2.381   0.773   2.433  1.00  0.00
ATOM      7  O           1      -3.560   1.422   4.419  1.00  0.00
ATOM      8  O           1      -2.963  -0.435   2.208  1.00  0.00
ATOM      9  C           1      -1.455  -0.012   0.432  1.00  0.00
ATOM     10  C           1      -1.293   0.575  -0.967  1.00  0.00
ATOM     11  C           1      -0.022   1.456  -0.953  1.00  0.00
ATOM     12  C           1      -0.156   2.668   0.002  1.00  0.00
ATOM     13  C           1      -2.790  -0.688   0.814  1.00  0.00
ATOM     14  C           1      -4.014  -0.102   0.081  1.00  0.00
ATOM     15  C           1      -2.532   1.317  -1.376  1.00  0.00
ATOM     16  C           1      -3.744   1.008  -0.897  1.00  0.00
ATOM     17  O           1      -4.929   0.387   1.031  1.00  0.00
ATOM     18  C           1      -0.232  -0.877   0.763  1.00  0.00
ATOM     19  C           1       1.068  -0.077   0.599  1.00  0.00
ATOM     20  N           1       1.127   0.599  -0.684  1.00  0.00
ATOM     21  C           1       2.414   1.228  -0.914  1.00  0.00
ATOM     22  H           1       2.664   1.980  -0.132  1.00  0.00
ATOM     23  H           1       3.214   0.453  -0.915  1.00  0.00
ATOM     24  H           1       2.440   1.715  -1.915  1.00  0.00
ATOM     25  H           1      -0.719   3.474  -0.525  1.00  0.00
ATOM     26  H           1       0.827   3.106   0.281  1.00  0.00
ATOM     27  H           1      -2.264   3.702   4.086  1.00  0.00
ATOM     28  H           1      -0.781   4.288   2.207  1.00  0.00
ATOM     29  H           1      -0.301  -1.274   1.804  1.00  0.00
ATOM     30  H           1      -0.218  -1.756   0.076  1.00  0.00
ATOM     31  H           1      -4.617   1.581  -1.255  1.00  0.00
ATOM     32  H           1      -2.429   2.128  -2.117  1.00  0.00
ATOM     33  H           1      -4.464   1.058   1.509  1.00  0.00
ATOM     34  H           1      -2.749  -1.794   0.681  1.00  0.00
ATOM     35  H           1       1.170   0.665   1.425  1.00  0.00
ATOM     36  H           1       1.928  -0.783   0.687  1.00  0.00
ATOM     37  H           1      -3.640   2.223   4.961  1.00  0.00
ATOM     38  H           1       0.111   1.848  -1.991  1.00  0.00
ATOM     39  H           1      -1.166  -0.251  -1.707  1.00  0.00
ATOM     40  H           1      -4.560  -0.908  -0.462  1.00  0.00
```

CONECT records: The CONECT records specify connectivity between atoms for which coordinates are supplied. The connectivity is described using the atom serial number as found in the entry.

Record Format:

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	"CONECT"	
7 - 11	Integer	serial	Atom serial number
12 - 16	Integer	serial	Serial number of bonded atom
17 - 21	Integer	serial	Serial number of bonded atom
22 - 26	Integer	serial	Serial number of bonded atom
27 - 31	Integer	serial	Serial number of bonded atom
32 - 36	Integer	serial	Serial number of hydrogen bonded atom
37 - 41	Integer	serial	Serial number of hydrogen bonded atom
42 - 46	Integer	serial	Serial number of salt bridged atom

47 - 51	Integer	serial	Serial number of hydrogen bonded atom
52 - 56	Integer	serial	Serial number of hydrogen bonded atom
57 - 61	Integer	serial	Serial number of salt bridged atom

Example:

```
          1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890
CONECT 1179  746 1184 1195 1203
CONECT 1179 1211 1222
CONECT 1021  544 1017 1020 1022 1211 1222      1311
```

As we noted at the beginning of this Appendix, PDB files can be extremely complex, and most of the examples we have found have been fairly large. The file shown in Figure 17.2 below is among the simplest PDB files we've seen, and describes the adrenalin molecule. This is among the materials provided as `adrenaline.pdb`.

```
HEADER    NONAME 08-Apr-99                                                NONE   1
TITLE                                                                     NONE   2
AUTHOR    Frank Oellien                                                   NONE   3
REVDAT   1  08-Apr-99     0                                               NONE   4
ATOM      1  C           0      -0.017   1.378   0.010  0.00  0.00           C+0
ATOM      2  C           0       0.002  -0.004   0.002  0.00  0.00           C+0
ATOM      3  C           0       1.211  -0.680  -0.013  0.00  0.00           C+0
ATOM      4  C           0       2.405   0.035  -0.021  0.00  0.00           C+0
ATOM      5  C           0       2.379   1.420  -0.013  0.00  0.00           C+0
ATOM      6  C           0       1.169   2.089   0.002  0.00  0.00           C+0
ATOM      7  O           0       3.594  -0.625  -0.035  0.00  0.00           O+0
ATOM      8  O           0       1.232  -2.040  -0.020  0.00  0.00           O+0
ATOM      9  C           0      -1.333   2.112   0.020  0.00  0.00           C+0
ATOM     10  O           0      -1.177   3.360   0.700  0.00  0.00           O+0
ATOM     11  C           0      -1.785   2.368  -1.419  0.00  0.00           C+0
ATOM     12  N           0      -3.068   3.084  -1.409  0.00  0.00           N+0
ATOM     13  C           0      -3.443   3.297  -2.813  0.00  0.00           C+0
ATOM     14  H           0      -0.926  -0.557   0.008  0.00  0.00           H+0
ATOM     15  H           0       3.304   1.978  -0.019  0.00  0.00           H+0
ATOM     16  H           0       1.150   3.169   0.008  0.00  0.00           H+0
ATOM     17  H           0       3.830  -0.755  -0.964  0.00  0.00           H+0
ATOM     18  H           0       1.227  -2.315  -0.947  0.00  0.00           H+0
ATOM     19  H           0      -2.081   1.509   0.534  0.00  0.00           H+0
ATOM     20  H           0      -0.508   3.861   0.214  0.00  0.00           H+0
ATOM     21  H           0      -1.037   2.972  -1.933  0.00  0.00           H+0
ATOM     22  H           0      -1.904   1.417  -1.938  0.00  0.00           H+0
ATOM     23  H           0      -3.750   2.451  -1.020  0.00  0.00           H+0
ATOM     24  H           0      -3.541   2.334  -3.314  0.00  0.00           H+0
ATOM     25  H           0      -4.394   3.828  -2.859  0.00  0.00           H+0
ATOM     26  H           0      -2.674   3.888  -3.309  0.00  0.00           H+0
CONECT    1    2    6    9    0                                           NONE  31
CONECT    2    1    3   14    0                                           NONE  32
CONECT    3    2    4    8    0                                           NONE  33
CONECT    4    3    5    7    0                                           NONE  34
CONECT    5    4    6   15    0                                           NONE  35
CONECT    6    5    1   16    0                                           NONE  36
CONECT    7    4   17    0    0                                           NONE  37
CONECT    8    3   18    0    0                                           NONE  38
CONECT    9    1   10   11   19                                           NONE  39
CONECT   10    9   20    0    0                                           NONE  40
CONECT   11    9   12   21   22                                           NONE  41
CONECT   12   11   13   23    0                                           NONE  42
CONECT   13   12   24   25   26                                           NONE  43
END                                                                       NONE  44
```

Figure 17.1: Example of a simple molecule file in PDB format

Appendix II: CTL file format

The structure of the CT file is straightforward. The file is segmented into several parts, including a header block, the counts line, the atom block, the bond block, and other information. The header block is the first three lines of the file and include the name of the molecule (line 1); the user's name, program, date, and other information (line 2); and comments (line 3). The next line of the file is the counts line and contains the number of molecules and the number of bonds as the first two entries. The next set of lines is the atom block that describes the properties of individual atoms in the molecule; each contains the X-, Y-, and Z-coordinate and the chemical symbol for an individual atom. The next set of lines is the bonds block that describes the properties of individual bonds in the molecule; each line contains the number (starting with 1) of the two atoms making up the bond and an indication of whether the bond is single, double, triple, etc. After these lines are more lines with additional descriptions of the molecule that we will not use for this project. An example of a simple CTfile-format file for a molecule (from the reference) is given in Figure 17.2 below.

Obviously there are many pieces of information in the file that are of interest to the chemist, and in fact this is an extremely simple example of a file. But for our project we are only interested in the geometry of the molecule, so the additional information in the file must be skipped when the file is read.

```
L-Alanine (13C)
GSMACCS-II10169115362D 1 0.00366 0.00000 0

6 5 0 0 1 0 3 V2000
-0.6622  0.5342 0.0000 C    0 0 2 0 0 0
 0.6220 -0.3000 0.0000 C    0 0 0 0 0 0
-0.7207  2.0817 0.0000 C    1 0 0 0 0 0
-1.8622 -0.3695 0.0000 N    0 3 0 0 0 0
 0.6220 -1.8037 0.0000 O    0 0 0 0 0 0
 1.9464  0.4244 0.0000 O    0 5 0 0 0 0
1 2 1 0 0 0
1 3 1 1 0 0
1 4 1 0 0 0
2 5 2 0 0 0
2 6 1 0 0 0
M CHG 2 4 1 6 -1
M ISO 1 3 13
M END
```

Figure 17.2: Example of a simple molecule file in CTfile format

Appendix III: the STL file format

The STL (sometimes called StL) file format is used to describe a file that contains information for 3D hardcopy systems. The name "STL" comes from stereo lithography, one of the technologies for 3D hardcopy, but the format is used in several other hardcopy technologies as described in the hardcopy chapter.

The .stl or stereolithography format describes an ASCII or binary file used in manufacturing. It is a list of the triangular surfaces that describe a computer generated solid model. This is the standard input for most rapid prototyping machines as described in the chapter of these notes on hardcopy. The binary format for the file is the most compact, but here we describe only the ASCII format because it is easier to understand and easier to generate as the output of student projects.

The ASCII .stl file must start with the lower case keyword `solid` and end with `endsolid`. Within these keywords are listings of individual triangles that define the faces of the solid model. Each individual triangle description defines a single normal vector directed away from the solid's surface followed by the xyz components for all three of the vertices. These values are all in Cartesian coordinates and are floating point values. The triangle values should all be positive and contained within the building volume. For this project the values are 0 to 14 inches in x, 0 to 10 in the y and 0 to 12 in the z. This is the maximum volume that can be built but the models should be scaled or rotated to optimize construction time, strength and scrap removal. The normal vector is a unit vector of length one based at the origin. If the normals are not included then most software will generate them using the right hand rule. If the normal information is not included then the three values should be set to 0.0. Below is a sample ASCII description of a single triangle within an STL file.

```
solid
...
facet normal 0.00 0.00 1.00
   outer loop
     vertex  2.00  2.00  0.00
     vertex -1.00  1.00  0.00
     vertex  0.00 -1.00  0.00
   endloop
 endfacet
...
endsolid
```

When the triangle coordinates are generated by a computer program,it is not unknown for roundoff errors to accumulate to the point where points that should be the same have slightly different coordinates. For example, if you were to calculate the points on a circle by incrementing the angle as you move around the circle, you might well end up with a final point that is slightly different from the initial point. File-checking software will note any difference between points and may well tell you that your object is not closed, but that same softward will often "heal" small gaps in objects automatically.

Vertex to vertex rule

The most common error in an STL file is non-compliance with the vertex-to-vertex rule. The STL specifications require that all adjacent triangles share two common vertices. This is illustrated in Figure 17.3. The figure on the left shows a top triangle containing a total of four vertex points. The outer vertices of the top triangle are not shared with one and only one other single triangle. The lower two triangles each contain one of the points as well as the fourth invalid vertex point.

To make this valid under the vertex to vertex rule the top triangle must be subdivided as in the example on the right.

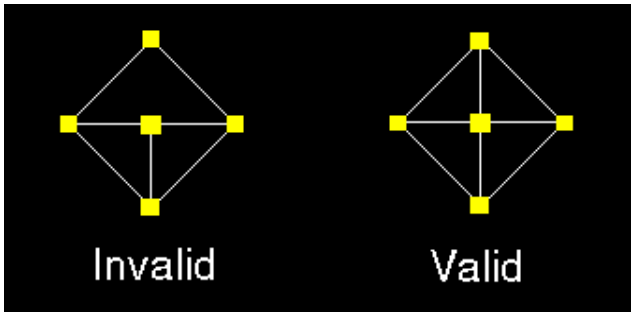


Figure 17.3

References:

CTFile Formats, MDL Information Systems, Inc., San Leandro, CA 94577, 1999. Available by download from `http://www.mdli.com/`

Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description, version 2.1, available online from `http://www.pdb.bnl.gov`

References and Resources

References

[AMES] Ames, Andrea L., David R. Nadeau, and John L. Moreland, *VRML 2.0 Sourcebook*, Wiley, 1997

[AN00] Angel, Ed, *Interactive Computer Graphics with OpenGL*, second edition, Addison-Wesley, 2000

[AN02] Angel, Ed, OpenGL: A Primer, Addison-Wesley, 2002

[BAK] Baker, Steve, A Brief MUI User Guide, distributed with the MUI release

[BAN] Banchoff, Tom et al., "Student-Generated Software for Differential Geometry," in Zimmermann and Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 165-171

[BRA] Braden, Bart, "A Vector Field Approach in Complex Analysis," in *Visualization in Teaching and Learning Mathematics*, Zimmermann and Cunningham, eds., MAA Notes Number 19, Mathematical Association of America, 1991, pp. 191–196

[BR95] Brown, Judith R. et al., *Visualization: Using Computer Graphics to Explore Data and Present Information*, Wiley, 1995

[BR99] K. A. Robson Brown, A. Chalmers, T. Saigol, C. Green, F.d'Errico, An automated laser scan survey of the Upper Palaeolithic rock shelter of Cap Blanc, *Journal of Archeological Science*, 1999

[BUC] Buchanan, J.L. et al., "Geometric Interpretation of Solutions in Differential Equations," in *Visualization in Teaching and Learning Mathematics*, Zimmermann and Cunningham, eds., MAA Notes Number 19, Mathematical Association of America, 1991, pp. 139–147

[CU92] Cunningham, S. and R. J. Hubbold, eds., Interactive Learning through Visualization, Springer-Verlag, 1992

[DU] Durett, H. John, ed., Color and the Computer, Academic Press, 1987

[EB] Ebert, David et al., *Texturing and Modeling: a Procedural Approach*, second edition, Academic Press, 1998

[FO] Foley, James D. et al, *Computer Graphics Principles and Practice*, 2nd edition, Addison-Wesley, 1990

[GL] Glassner, Andrew S., ed., An Introduction to Ray Tracing, Academic Press, 1989

[HIL] Hill, F. S. Jr., *Computer Graphics Using OpenGL*, 2nd edition, Prentice-Hall, 2001

[LA] Landau, Rubin H. and Manuel J. Páez, Computational Physics: Problem Solving with Computers, Wiley, 1997

[MUR] Murray, James D. and William vanRyper, *Encyclopedia of Graphics File Formats*, second edition, O'Reilly & Associates, 1996

[PER] Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3), Proceedings of SIGGRAPH 85, July 1985, 287-296

[PIE] Pietgen, Heinz-Otto and Dietmar Saupe, eds., *The Science of Fractal Images*, Springer-Verlag, New York, 1988

[POR] Porter and Duff, "Compositing Digital Images," *Proceedings*, SIGGRAPH 84 Conference, ACM SIGGRAPH, 1984

[SHR] Shreiner, Dave, ed., *OpenGL Reference Manual*, 3rd edition, Addison-Wesley, 2000

[SOW1] Sowrizal, Henry, Kevin Rushforth, and Michael Deering, *The Java3D™ 3D API Specification*, Addison-Wesley, 1995

[SOW2] Sowizral, Henry A. and David R. Nadeau, *Introduction to Programming with Java 3D*, SIGGRAPH 99 Course Notes, Course 40

[SPE] Spence, Robert, Information Visualization, Addison-Wesley/ACM Press Books, 2001

[SVR] The SIGGRAPH Video Review (SVR), an excellent source of animations for anyone wanting to see how images can communicate scientific and cultural information through computer graphics, as well as how computer graphics can be used for other purposes. Information on SVR can be found at http://www.siggraph.org/SVR/

[TALL] Tall, David, "Intuition and Rigour: The Role of Visualization in the Calculus," in *Visualization in Teaching and Learning Mathematics*, Zimmermann and Cunningham, eds., MAA Notes Number 19, Mathematical Association of America, 1991, pp. 105–119

[THO] Thorell, R. G. and W. J. Smith, Using Computer Color Effectively: An Illustrated Reference, Prentice-Hall, 1990

[UP] Upstill, Steve, The RenderMan™ Companion, Addison-Wesley, 1990

[ViSC] McCormick, Bruce H., Thomas A. DeFanti, and Maxine D. Brown, eds., *Visualization in Scientific Computing*, *Computer Graphics* 21(6), November 1987

[vSEG] von Seggern, David, *CRC Standard Curves and Surfaces*, CRC Press, 1993

[WAT] Watt, Alan and Fabio Policarpo, 3D Games: Real-time Rendering and Software Technology, Addison-Wesley/ACM SIGGRAPH Series, 2001

[WA2] Watt, Alan and Mark Watt, Advanced Animation and Rendering Techniques: Theory and Practice, Addison-Wesley, 1992

[WO98] Wolfe, Rosalee, ed., Seminal Graphics: Pioneering Efforts that Shaped the Field, ACM SIGGRAPH, 1998

[WO00] Wolfe, R. J., *3D Graphics: A Visual Approach*, Oxford University Press, 2000

[WOO] Woo, Mason et al., *OpenGL Programmers Guide*, 3rd edition (version 1.2), Addison-Wesley, 1999

[ZIM] Zimmermann, Walter and Steve Cunningham, *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991

- Textbook for multivariate calculus ...
- Textbook for differential geometry? ...
- Textbook or reference for simulations

Resources

Chromadepth information is available from

Chromatek Inc
1246 Old Alpharetta Road
Alpharetta, GA 30005
888-669-8233
http://www.chromatek.com/

Chromadepth glasses may be ordered from

American Paper Optics
3080 Bartlett Corporate Drive
Bartlett, TN 38133
800-767-8427, 901-381-1515
fax 901-381-1517

Below are contacts for 3D hardcopy sources:

3D Systems
26081 Avenue Hall
Valencia, CA 91355 USA
+1.888.337.9786 (USA toll-free)
moreinfo@3dsystems.com
http://www.3dsystems.com

Helisys, Inc.
1000 E. Dominguez Street
Carson, CA 90746-3608 USA
+1.310.630.8840
http://helisys.com/

Solidscape, Inc
316 Daniel Webster Highway
Merrimack, NH 03054-4115 USA
+1-603-429-9700; fax: 603-424-1850
e-mail: precision@solid-scape.com
http://www.solid-scape.com/

Stratasys, Inc.
14950 Martin Drive
Eden Prairie, MN 55344-2020 USA
+1-888-480-3548, +1.952.937.3000
fax: +1.952.937.0070
e-mail: info@stratasys.com
http://www.stratasys.com/

z corporation
20 North Avenue
Burlington, MA 01803 USA
+1.781-852-5005
http://www.zcorp.com/

Evaluation

These notes are under development, and we are very interested in hearing from anyone who uses them. Following this cover page we have added pages with questions for both instructors and students. As we work to develop the notes, your answers and comments will help us identify areas where we can improve the notes and will let us consider your ideas on how to make those improvements.

You can send us your comments by mail, by email, or by fax, and you may tell us who you are or remain anonymous (within the limits of the way you contact us, of course). If you respond by email, please send your comments to rsc@cs.csustan.edu. If you respond by post, please send your comments to

Steve Cunningham
Computer Science Department
California State University Stanislaus
801 W. Monte Vista Avenue
Turlock, CA 95382 USA

All comments will be acknowledged and will be fully considered.

Instructor's evaluation

1. How did you use these notes? Did you review them for a course or simply to evaluate their possible use? Did you decide to use them for a course or did you decide not to use them?
2. If you chose to use them for a course, what was it about the notes that led you to that choice? If you chose not to use them, what was it about the notes that led you to that choice?
3. If you chose to use them, were they used as a course text, a course supplement, a student resource, or your personal resource? Were they used in a regular course, a special topics course, or a readings course?
4. If you chose to use them, how did your students respond to the notes?

While the notes are clearly incomplete and under development, we want your comments on the content. We would remind you of the goals of the notes as presented in the Getting Started chapter as you discuss the content.

5. Do you find the goals of the notes to represent a worthwhile approach to the introductory computer graphics course? Whether yes or no — but especially if no — we would value your feedback on these goals.
6. Were there any topics in the notes that seemed superfluous and could be omitted without any effect on your course?
7. Do you agree with the choice of OpenGL as the API for these notes, or do you suggest another API? Should the notes emphasize general concepts first in each section and then discuss the OpenGL implementation of the concepts, or should they use OpenGL as a motivator of the general discussion throughout?
8. Was the sequence of topics in the notes appropriate, or did you find that you would need to teach them in a different order to cover the material effectively?
9. Were there any topics in the notes that seemed particularly valuable to your course? Should these be emphasized, either through expanding their presence or through highlighting them in other parts of the notes?
10. Are the notes accurate? Is there any place where the notes are incorrect or misleading (not all areas have been fully tested, so this is possible)?
11. Are there areas where the discussions are difficult for students to follow?
12. Would you want to have supplementary material to accompany the notes? What kind of things would you want in such material? Should that material be on an accompanying CD-ROM or on an archival Web site?
13. Is there anything else — positive or negative — you would want to tell the author and the development group for this project?
14. Please tell us a bit about yourself: your department, your teaching and research interests, your resons for being interested in directions in computer graphics instruction, and anything else that will help us understand your comments above.

Thank you very much for your assistance.

Student's evaluation

1. How did you find out about these notes? Was it from your instructor, from a friend, or from a general search of online computer graphics resources?

2. How did you use these notes? Was it in a class or as a personal resource?

3. If you used the notes in a class, what kind of class was it? Was it a beginning computer graphics course or did you have another computer graphics course first? Was it a regular undergraduate course, a special topics course, a readings course, or another kind of course? What department was the course offered in?

4. Do you agree with the use of OpenGL as the graphics API for the course that used these notes? Would you rather have had a different API? If so, what one and why?

5. Whak kind of system support did you use for the course (Windows, Linux, Unix, Macintosh, etc.)? Did you have to install any extra features (GLUT, MUI, etc.) yourself to use the notes? Did you need any extra instruction in the use of your local systems in order to use the ideas in the notes?

6. Without considering other important aspects of your course (laboratory, instructor, etc.), did you find the notes a helpful resource in learning computer graphics? Were you able to follow the discussions and make sense of the code fragments?

7. Were there any topics in the notes that seemed particularly valuable to your reading or your course? Should these be emphasized, either through expanding their presence or through highlighting them in other parts of the notes?

8. Were there any topics in the notes that seemed superfluous and could be removed without hurting your learning?

9. Were there any topics in computer graphics that you wanted to see in your reading or course that the notes did not cover? Why were these important to you?

10. Would you have liked to have additional materials to go along with the notes? What would you have wanted? How would you like to get these materials: on CD-ROM, on a Web site, or some other way?

11. Is there anything else — positive or negative — you would want to tell the author and the development group for this project?

12. Please tell us a bit about yourself: your major and year, your particular interests in computing and in computer graphics, your career goals, and anything else that will help us understand your comments above.

Thank you very much for your assistance.