Contents

APPENDIX
B

NUMERICAL METHODS 469

List of Figures

List of Tables

PREFACE

This book is the culmination of many years of reading and participating in the Internet newsgroups on computer graphics and computer games, most notably *comp.graphics.algorithms* and the hierarchy of groups *comp.games.development.**. The focus of my participation has been to provide free source code that solves common problems that arise in computer graphics, image analysis, and numerical methods, available through Magic Software at *www.magic-software.com*. The book is also a technical summary of my experiences in helping to produce a commercial game engine, NetImmerse, developed by Numerical Design Limited (NDL), *www.ndl.com*.

The focus of this book is on understanding that a game engine, or more generally a real-time computer graphics engine, is a complex entity that consists of more than simply a rendering layer that draws triangles. It is also more than just a collection of unorganized techniques. A game engine must deal with issues of scene graph management as a front end that efficiently provides the input to the back end renderer, whether it be a software- or hardware-based renderer. The engine must also provide the ability to process complex and moving objects in a physically realistic way. The engine must support collision detection, curved surfaces as well as polygonal models, animation of characters, geometric level of detail, terrain management, and spatial sorting. Moreover, the engine is large enough that the principles of object-oriented design must be practiced with great care.

The chapters of this book tend to be fairly mathematical and geometrical. The intended audience includes anyone who is interested in becoming involved in the development of a real-time computer graphics engine. It is assumed that the reader's background includes a basic understanding of vector and matrix algebra, linear algebra, multivariate calculus, and data structures.

Many people have directly or indirectly contributed to the book. Most notable are the engineers at NDL: Lars Bishop, Jon McAllister, Chad Robertson, Rob Phillips, Tim Preston, Scott Sherman, Ed Holzworth, and Andy Jones. Lars and I are the primary architects for NetImmerse. He is the renderer expert, especially with regards to Direct3D, and has been instrumental in helping me to understand many of the issues for rendering. We also have had many productive design sessions about how best to incorporate the ideas for scene graph management to properly feed the renderers and to properly manage renderer state. Chad and Rob are the animation experts. They did a lot of legwork on understanding how various modeling packages animate characters and deciding how NetImmerse can best support the animation. Chad also contributed many good ideas on how to structure the collision detection system to work well with the hierarchical scene graph system. Jon is the expert on continuous level of detail and has implemented some of the algorithms mentioned in this book for NetImmerse. The implementations go well beyond what is discussed here and

addressed practical concerns that some of the research papers did not cover. Jon also worked with Chad and Rob on the integration of continuous level of detail with the skin-and-bones system, a nontrivial task. Tim was helpful in reading Chapter 8 and attempting to implement the top-down algorithm as I originally wrote it. He pointed out what I had overlooked, leading to some fine discussions about how to properly tessellate the surfaces without paying for a large memory overhead. The algorithm as described in this book reflects these discussions. Finally, Bill Baxter was a summer intern from the University of North Carolina, but in his time at NDL was able to investigate the topic of inverse kinematics and implement that system in NetImmerse. Discussions with him led to my understanding of how inverse kinematics should work in the game engine and is reflected in how I wrote the section on that topic.

I want to thank the reviewers for the book: Ian Ashdown (byHeart Consultants Limited), John Laird (University of Michigan), Jeff Lander (Darwin 3D), Franz Lanzinger (Actual Entertainment), Peter Lipson (Mindscape), Tomas Möller (Chalmers), Andrea Pessino (Blizzard Entertainment), and Steve Woodcock (Raytheon). They spent a quite large amount of time reading over the two drafts of the book and provided many helpful comments and criticisms. I also want to thank my editor, Tim Cox, and his assistant, Brenda Modliszewski, for the time they have put into helping the book come to completion.

CHAPTER 1

INTRODUCTION

I have no fault to find with those who teach geometry. That science is the only one which has not produced sects; it is founded on analysis and on synthesis and on the calculus; it does not occupy itself with probable truth; moreover it has the same method in every country.

— Frederick the Great

1.1 A BRIEF MOTIVATION

Computer graphics has been a popular area of computer science for the last few decades. Much of the research has been focused on obtaining physical realism in rendered images, but generating realistic images comes at a price. The algorithms tend to be computationally expensive and must be implemented on high-end, special-purpose graphics hardware affordable only by universities through research funding or by companies whose focus is computer graphics. Although computer games have also been popular for decades, for most of that time the personal computers available to the general public have not been powerful enough to produce realistic images. The game designers and programmers have had to be creative to produce immersive environments that draw the attention of the player to the details of game play and yet do not detract from the game by the low-quality graphics required for running on a low-end machine.

Times are changing. As computer technology has improved, the demand for more realistic computer games that support real-time interaction has increased. Moreover, the group of computer gamers itself has evolved from a small number of, shall we say, computer geeks to a very large segment of the population. One of the most popular, successful, and best-selling games was *Myst*, created and produced by Cyan Productions and published through Broderbund. This game and others like it showed that an entirely new market was possible—a market that included the general consumer, not just computer-savvy people. The increased demand for games and the potential size of the market has created an impetus for increased improvement in the computer technology—a not-so-vicious circle.

One result of the increased demand has been the advent of hardware-accelerated graphics cards that off-load a lot of the work a CPU normally does for software rendering. The initial cards were add-ons that handled only the 3D acceleration and ran only in full-screen mode. The 2D graphics cards were still used for the standard graphics display interface (GDI) calls. Later-generation accelerators have been designed to handle both 2D GDI and 3D acceleration within a window that is not full screen. Since triangle rasterization has been the major bottleneck in software rendering, the hardware-accelerated cards have acted as fast triangle rasterizers. As of the time of this writing, the next-generation hardware cards are being designed to off-load even more work. In particular, the cards will perform point transformations and lighting calculations in hardware.

Another result of the increased demand for games has been the evolution of the CPUs themselves to include support for operations that typically arise in game applications: fast division, fast inverse square roots (for normalizing vectors), and parallelism to help with transforming points and computing dot products. The possibilities for the evolutionary paths are endless. Many companies are now exploring new ways to use the 3D technology in applications other than games, for example, in Web commerce and in plug-ins for business applications.

And yet one more result of the increased demand is that a lot of people now want to write computer games. The Internet newsgroups related to computer graphics, computer games, and rendering application programmer interfaces (APIs) are filled with questions from eager people wanting to know how to program for games. At its highest level, developing a computer game consists of a number of factors. First and foremost (at least in my opinion) is having a good story line and good game play—without this, everything else is irrelevant. Creation of the story line and deciding what the game play should be can be categorized as *game design.* Once mapped out, artists must build the *game content,* typically through modeling packages. Interaction with the content during run time is controlled through *game artificial intelligence*, more commonly called *game AI*. Finally, programmers must create the application to load content when needed, integrate the AI to support the story line and game play, and build the *game engine* that manages the data in the world and renders it on the computer screen. The last topic is what this book is about—building a sophisticated real-time game engine. Although games certainly benefit from real-time computer

graphics, the ideas in this book are equally applicable to any other area with three-dimensional data, such as scientific visualization, computer-aided design, and medical image analysis.

1.2 A Summary of the Chapters

The classical view of what a computer graphics engine does is the *rendering* of triangles (or polygons). Certainly this is a necessary component, but it is only half the story. Viewed as a black box, a renderer is a consumer-producer. It consumes triangles and produces output on a graphics raster display. As a consumer it can be fed too much data, too quickly, or it can be starved and sit idly while waiting for something to do. A front-end system is required to control the input data to the renderer; this process is called *scene graph management.* The main function of the scene graph management is to provide triangles to the renderer, but how those triangles are obtained in the first place is a key aspect of the front end. The more realistic the objects in the scene, the more complex the process of deciding which triangles are sent to the renderer. Scene graph management consists of various modules, each designed to handle a particular type of object in the world or to handle a particular type of process. The common theme in most of the modules is *geometry*.

Chapter 2 covers basic background material on geometrical methods, including matrix transformations, coordinate systems, quaternions, Euler angles, the standard three-dimensional objects that occur most frequently when dealing with bounding volumes, and a collection of distance calculation methods.

The graphics pipeline, the subject of Chapter 3, is discussed in textbooks on computer graphics to varying degrees. Some people would argue against the inclusion of some parts of this chapter, most notably the sections on rasterization, contending that hardware-accelerated graphics cards handle the rasterization for you, so why bother expounding on the topic. My argument for including these sections is twofold. First, the computer games industry has been evolving in a way that makes it difficult for the "garage shop" companies to succeed. Companies that used to focus on creating games in-house are now becoming publishers and distributors for other companies. If you have enough programmers and resources, there is a chance you can convince a publisher to support your effort. However, publishers tend to think about reaching the largest possible market and often insist that games produced by their clients run on low-end machines without accelerated graphics cards. And so the clients, interested in purchasing a third-party game engine, request that software renderers and rasterizers be included in the package. I hope this trend goes the other way, but the commercial reality is that it will not, at least in the near future. Second, hardware-accelerated cards do perform rasterization, but hardware requires drivers that implement the high-level graphics algorithms on the hardware. The cards are evolving rapidly, and the quality of the drivers is devolving at the same rate—no one wants to fix bugs in the drivers

for a card that will soon be obsolete. But another reason for poor driver quality is that programming 3D hardware is a much more difficult task than programming 2D hardware. The driver writers need to understand the hardware and the graphics pipeline. This chapter may be quite useful to that group of programmers.

Chapter 4 introduces scene graph management and provides the foundation for a hierarchical organization designed to feed the renderer efficiently, whether a software or hardware renderer. The basic concepts of local and world transforms, bounding volumes for culling, render state management, and animation support are covered.

Chapters 5 and 6 discuss aspects of the intersection of objects in the world. Picking is the process of computing the intersection of a line, ray, or line segment with objects. Collision detection refers to computing intersections between planar or volumetric objects. Some people include picking as part of the definition of collision detection, but the complexity of collision systems for nonlinear objects greatly exceeds that for picking, so I have chosen to separate the two systems.

Chapters 7 through 12 cover various systems that are supported by the scene graph management system. Chapters 7 and 8, on curves and surfaces, are somewhat general, but the emphasis is on tessellation. The next-generation game consoles have powerful processors but are limited in memory and bandwidth between processors. The dynamic tessellation of surfaces is desirable since the surfaces can be modeled with a small number of control points (reducing memory usage and bandwidth requirements) and tessellated to as fine a level as the processors have cycles to spare. The emphasis will start to shift from building polygonal models to building curved surface models to support the trend in new hardware on game consoles. Chapter 9 discusses the animation of geometric data, and in particular, key frame animation, inverse kinematics, and skin-and-bones systems. Level of detail is the subject of Chapter 10, with a special focus on continuous level of detail, which supports dynamic change in the number of triangles to render based on view frustum parameters.

Chapter 11 presents an algorithm for handling terrain. Although there are other algorithms that are equally viable, I chose to focus on one in detail rather than briefly talk about many algorithms. The key ideas in implementing this terrain algorithm are applicable to implementing other algorithms. High-level sorting algorithms, including portals and binary space partitioning trees, are the topic of Chapter 12.

Chapter 13 provides a brief survey of special effects that can be used in a game engine. The list is not exhaustive, but it does give an idea of what effects are possible with not much effort.

Building a commercial game engine certainly requires understanding a lot about computer graphics, geometry, mathematics, and data structures. Just as important is properly architecting the modules so that they all integrate in an efficient manner. A game engine is a large library to which the principles of object-oriented design apply. Appendix A provides a brief review of those principles and includes a discussion on an object-oriented infrastructure that makes maintenance of the library easier down the road. These aspects of building an engine are often ignored because it is faster and easier to try to get the basic engine up and running right away. However, short-

term satisfaction will inevitably come at the price of long-term pain in maintenance. Appendix B is a summary of various numerical methods that, in my experience, are necessary to implement the modules described in Chapters 7 through 12.

1.3 Text Is Not Enough

This book is not like the academic textbooks you would find in the school bookstore or the popular computer game programming books that you see at your favorite bookseller. Academic texts on computer graphics tend to be tomes covering a large number of general topics and are designed for learning the basic concepts, not for implementing a full-blown system. Algorithmic details are modest in some books and lacking in others. The popular programming books present the basic mathematics and concepts, but in no way indicate how complex a process it is to build a good engine. The technical level in those books is simply insufficient.

A good collection of books that address more of the algorithmic issues for computer graphics is the *Graphics Gems* series (Glassner 1990; Aarvo 1991; Kirk 1992; Heckbert 1994; Paeth 1995). Although providing a decent set of algorithms, the collection consists of contributions from various people with no guidance as to how to incorporate these into a larger integrated package such as a game engine. The first real attempt at providing a comprehensive coverage of the topics required for real-time rendering is Möller and Haines (1999), which provides much more in-depth coverage about the computer graphics topics relevant to a real-time graphics engine. The excellent references provided in that book are a way to investigate the roots of many of the concepts that current-generation game engines incorporate.

But there is one last gap to fill. Textual descriptions of graphics algorithms, no matter how detailed, are difficult to translate into real working code, even for experienced programmers. Just try to implement some of the algorithms described in the ACM SIGGRAPH proceedings! Many of those articles were written after the authors had already worked out the details of the algorithms and implemented them. That process is not linear. Ideas are formulated, algorithms are designed, then implemented. When the results of the coding point out a problem with the algorithmic formulation, the ideas and algorithms are reformulated. This natural process iterates until the final results are acceptable. Written and published descriptions of the algorithms are the final summary of the final algorithm. However, taken out of context of the idea-to-code environment, they sometimes are just not enough. Because having an actual implementation to look at while attempting to learn the ideas can only accelerate the learning process, a CD-ROM containing an implementation of a game engine accompanies this book. While neither as feature complete nor as optimized as a commercial engine, the code should help in understanding the ideas and how they are implemented. Pointers to the relevant source code files that implement the ideas are given in the text.

CHAPTER 4

HIERARCHICAL SCENE REPRESENTATIONS

SOURCE CODE

LIBRARY
Engine

FILENAME
All Files

The graphics pipeline discussed in Chapter 3 requires that each drawable object be tested for culling against the view frustum and, if not culled, be passed to the renderer for clipping, lighting, and rasterizing. Given a 3D world with a large number of objects, the simplest method for processing the objects is to group them into a list and iterate over the items in the list for culling and rendering. Although this approach may be simple, it is not efficient since each drawable object in the world must be tested for culling.

A better method for processing the objects is to group them hierarchically according to spatial location. The grouping structure discussed in this chapter is a *tree*. The tree has leaf nodes that contain geometric data and internal nodes that provide a grouping mechanism. Each node has one parent (except for the root node, which has none) and any number of child nodes. It is possible to use a directed acyclic graph as an attempt to support high-level sharing of objects. Each node in the graph can have multiple parents, each parent sharing the object represented by the subgraph rooted at the node. However, the memory costs and code complexity to maintain such a graph do not justify using it. Sharing should occur at a lower level so that leaf nodes can

share vertices, texture images, and other data that tends to use a lot of memory. The implied links from sharing are not part of the parent-child relationships in the hierarchy. Regardless of whether trees or directed acyclic graphs are used, the resulting set of grouped objects is called a *scene graph*.

The organization of content in a scene graph is quite important for games in many ways, of which four are listed here. First, the amount of content to manage is typically large and is built in small pieces by the artists. The level editor can assemble the content for a single level as a hierarchy by concentrating on the local items of interest. The global ramifications are effectively the responsibility of the hierarchy itself. For example, a light in the world can be chosen to illuminate only a subtree of the graph. The level editor's responsibility is to assign that light to a node in the graph. The effect of the light on the subtree rooted at that node is automatically handled by the scene graph management system. Second, hierarchical organization provides a form of locality of reference, a common concept in memory management by a computer system. Objects that are of current interest in the game tend to occur in the same spatial region. The scene graph allows the game program to quickly eliminate other regions from consideration for further processing. Although minimizing the data sent to the renderer is an obvious goal to keep the game running fast, focusing on a small amount of data is particularly important in the context of collision detection. The collision system can become quite slow when the number of potentially colliding objects is large. A hierarchical scene graph supports grouping only a small number of potentially colliding objects, those objects occurring only in the local region of interest in the game. Third, many objects are naturally modeled with a hierarchy, most notably humanoid characters. The location and orientation of the hand of a character is naturally dependent on the locations and orientations of the wrist, elbow, and shoulder. Fourth, invariably the game must deal with persistence issues. A player wants to save the current game, and the game is to be continued at a later time. Hierarchical organization makes it quite simple to save the state of the world by asking the root node of the scene graph to save itself, the descendants saving themselves in a naturally recursive fashion.

Section 4.1 provides the basic concepts for management of a tree-based representation of a scene, including specification and composition of local and world transforms, construction of bounding volumes for use both in rapid view frustum culling and fast determination of nonintersection of objects managed by a collision system, selection and scope of renderer state at internal or leaf nodes, and control of animated quantities.

Changes in the world environment of the game are handled by changing various attributes at the nodes of the tree. A change at a single node affects the subtree for which that node is the root. Therefore, all nodes in the subtree must be notified of the change so that appropriate action can be taken. One typical action that requires an update of the scene graph is moving an object by changing its local transform. The world transforms of the object's descendants in the tree must be recalculated. Additionally, the object's bounding volume has changed, in turn affecting all the bounding volumes of its ancestors in the tree. The new bounding volume at a node

involves computing a single bounding volume that contains all the bounding volumes of its children, a process called *merging*. Another typical action that requires an update of the scene graph is changing renderer state at a node. The renderer state at all the leaf nodes in the affected tree must be updated. The update process is the topic of Section 4.2.

After a scene graph is updated, it is ready for processing by the renderer. The drawing pass uses the bounding volumes to cull entire subtrees at once, thereby reducing the amount of time the renderer has to spend on low-level processing of objects that ultimately will not appear on the computer screen. Section 4.3 presents culling algorithms for various bounding volumes compared to a plane at a time in the view frustum. The general drawing algorithm for a hierarchy is also discussed.

4.1 Tree-Based Representation

A simple grouping structure for objects in the world is a tree. Each node in the tree has exactly one parent, except for the root node, which has none. The root is the first node to be processed when attempting to render objects in the tree. The simplest example of a tree is illustrated in Figure 4.1. The top-level node is a *grouping node* (bicycle) and acts as a *parent* for the two *child nodes* (wheels). The children are grouped because they are part of the same object both spatially and semantically.

To take advantage of this structure, the nodes must maintain spatial and semantic information about the objects they represent. The main categories of information are *transforms*, *bounding volumes*, *render state*, and *animation state*. Transforms are used to position, orient, and size the objects in the hierarchy. Bounding volumes are used for hierarchical culling purposes and intersection testing. Render state is used to set up the renderer to properly draw the objects. Animation state is used to represent any time-varying node data.

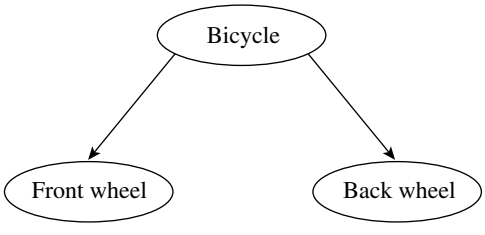

Figure 4.1 A simple tree with one grouping node.

4.1.1 TRANSFORMS

In Figure 4.1, it is not enough to know the semantic information that the two wheels are part of the bicycle. The spatial information, the location of the wheels, must also be specified. Moreover, it is necessary to know a coordinate system in which to specify that information. The parent node has its own coordinate system, and the location of a child is given relative to its parent's coordinates.

Local Transforms

The location of a node relative to its parent is represented abstractly as a homogeneous matrix with no perspective component. The matrix, called a *local transform,* represents any translation, rotation, scaling, and shearing of the node within the parent's coordinate system. While an implementation of scene graph nodes could directly store the homogeneous matrix as a 4×4 array, it is not recommended. The last row of the matrix is always $[0\ 0\ 0\ 1]$. Less memory is used if the homogeneous matrix is stored as a 3×3 matrix representing the upper-left block and a 3×1 vector representing the translation component of the matrix. This also avoids the inefficient general multiplication of homogeneous matrices and vectors since in that multiplication, there would be three multiplies by 0 and one multiply by 1. Given a homogeneous matrix with no perspective component, the matrix is denoted by

$$\left\langle M \;\middle|\; \vec{T} \right\rangle := \left[\begin{array}{c|c} M & \vec{T} \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]. \tag{4.1}$$

Using this compressed notation, the product of two homogeneous matrices is

$$\left\langle M_1 \;\middle|\; \vec{T}_1 \right\rangle \left\langle M_2 \;\middle|\; \vec{T}_2 \right\rangle = \left\langle M_1 M_2 \;\middle|\; M_1 \vec{T}_2 + \vec{T}_1 \right\rangle \tag{4.2}$$

and the product of a homogeneous matrix with a homogeneous vector $[\vec{V}|1]^{\mathrm{T}}$ is

$$\left\langle M \;\middle|\; \vec{T} \right\rangle \; \vec{V} = M\vec{V} + \vec{T}. \tag{4.3}$$

To keep the update time to a minimum and to avoid using numerical inversion of matrices in various settings, it is better to require that the local transform have only translation, rotation, and uniform scaling components. The general form of such a matrix is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle \tag{4.4}$$

and is called an *SRT-transform*. The uniform scaling factor is $s > 0$, the rotational component is the orthogonal matrix R whose determinant is one, and the translational component is $\vec{T}$. The product of two *SRT*-transforms is

$$\left\langle s_1 R_1 \;\middle|\; \vec{T}_1 \right\rangle \left\langle s_2 R_2 \;\middle|\; \vec{T}_2 \right\rangle = \left\langle s_1 s_2 R_1 R_2 \;\middle|\; s_1 R_1 \vec{T}_2 + \vec{T}_1 \right\rangle, \tag{4.5}$$

the product of an *SRT*-transform and a vector $\vec{V}$ is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle \vec{V} = sR\vec{V} + \vec{T}, \tag{4.6}$$

and the inverse of an *SRT*-transform is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle^{-1} = \left\langle \frac{1}{s} R^{\mathrm{T}} \;\middle|\; -\frac{1}{s} R^{\mathrm{T}} \vec{T} \right\rangle. \tag{4.7}$$

World Transforms

The local transform at a node specifies how the node is positioned with respect to its parent. The entire scene graph represents the world itself. The world location of the node depends on all the local transforms of the node and its predecessors in the scene graph. Given a parent node P with child node C, the *world transform* of C is the product of P's world transform with C's local transform,

$$\begin{aligned} \left\langle M_{\text{world}}^{(C)} \;\middle|\; \vec{T}_{\text{world}}^{(C)} \right\rangle &= \left\langle M_{\text{world}}^{(P)} \;\middle|\; \vec{T}_{\text{world}}^{(P)} \right\rangle \left\langle M_{\text{local}}^{(C)} \;\middle|\; \vec{T}_{\text{local}}^{(C)} \right\rangle \\ &= \left\langle M_{\text{world}}^{(P)} M_{\text{local}}^{(C)} \;\middle|\; M_{\text{world}}^{(P)} \vec{T}_{\text{local}}^{(C)} + \vec{T}_{\text{world}}^{(P)} \right\rangle. \end{aligned}$$

The world transform of the root node in the scene graph is just its local transform. The world position of a node N_k in a path $N_0 \cdots N_k$, where N_0 is the root node, is generated recursively by the above definition as

$$\left\langle M_{\text{world}}^{(N_k)} \;\middle|\; \vec{T}_{\text{world}}^{(N_k)} \right\rangle = \left\langle M_{\text{local}}^{(N_0)} \;\middle|\; \vec{T}_{\text{local}}^{(N_0)} \right\rangle \cdots \left\langle M_{\text{local}}^{(N_k)} \;\middle|\; \vec{T}_{\text{local}}^{(N_k)} \right\rangle.$$

4.1.2 Bounding Volumes

Object-based culling within a scene graph is very efficient whenever the bounding volumes of the nodes are properly nested. If the bounding volume of the parent node encloses the bounding volumes of the child nodes, culling of entire subtrees is supported. If the bounding volume of the parent node is outside the view frustum, then

the child nodes must be outside the view frustum and no culling tests need be done on the children. Hierarchical culling provides a fast way for eliminating large portions of the world from being processed by the renderer. The same nested bounding volumes support collision detection. If the bounding volume of the parent node does not intersect an object of interest, then neither do the child nodes. Hierarchical collision detection provides a fast way for determining that two objects do not intersect. The bounding volumes that are discussed in this chapter include spheres, oriented boxes, capsules, lozenges, cylinders, and ellipsoids.

A leaf node containing geometric data will also contain a bounding volume based on the model space coordinates of the data. However, the leaf node has a world space representation based on the product of local transforms from scene graph root to that leaf. That means the leaf node must also contain a world bounding volume, obtained by applying the world transform to the model bounding volume.

To support the efficiencies of a hierarchical organization of the world, an internal node requires a world bounding volume that contains the world bounding volumes of all its children. It is not necessary to maintain a model bounding volume at an internal node since such a node does not contain its own geometric data. While transforms are propagated from the root of the scene graph toward the leaf nodes, the bounding sphere calculations must occur from leaf node to root. A parent bounding volume cannot be known until its child bounding volumes are known. A recursive traversal downward allows computation of the world transforms. The upward return from the traversal allows computation of the world bounding volumes.

4.1.3 Renderer State

Renderer state can also be maintained in a hierarchical fashion. For example, if a subtree rooted at a node has all leaf nodes that want their textures to be alpha blended, the node can be tagged with state information that indicates alpha blending should be enabled for the entire subtree. Alternatively, tagging all the leaf nodes with the same renderer state information is an efficient use of memory. A traversal along a single path in the tree from root to leaf node accumulates the renderer state necessary to draw the geometry of the leaf node. Just before a leaf node is about to be drawn, the renderer processes the state information at that node and decides whether or not it needs to change its own internal state. As changes in rendering state can be expensive, the number of changes should be minimal. A typical expensive change involves using different textures. If a texture is in system memory but not in video memory, the texture must be copied to video memory, and that takes time. For sorting purposes, it is convenient to allow each leaf node to store a copy of the renderer state. A sorter can select a renderer state for which it wants to minimize changes, then sort the leaf nodes accordingly.

4.1.4 Animation

Animation in the classic sense is the motion of articulated characters and objects in the scene. If a character is represented hierarchically, each node might represent a joint (neck, shoulder, elbow, wrist, knee, etc.) whose local transformations change over time. Moreover, the values of the transformations are usually controlled by procedural means (see Chapter 9) as compared to the application manually adjusting the transforms. This can be accomplished by allowing each node to store *controllers,* with each controller managing some quantity that changes over time. In the case of classic animation, a controller might represent the local transform as a matrix function of time. For each specified time in the application, the matrix is computed by the controller and the world transform is computed using this matrix.

It is possible to allow any quantity at a node to change over time. For example, a node might be tagged to indicate that fogging is to be used in its subtree. The fog depth can be made to vary with time. A controller can be used to procedurally compute the depth based on current time. In this way animation is controlling any time-varying quantity in a scene graph.

4.2 Updating a Scene Graph

The scene graph represents the state of the world at a given time. If the state changes for whatever reason, the scene graph must be updated to represent the new state. Typical state changes include model data changing at a node, local transforms changing at a node, the topological structure of the tree changing, renderer state changing, or some animated quantity changing. Updating the scene graph is only necessary in those subtrees affected by the changes. For example, if a local transform is changed at a single node, then only the subtree rooted at that node is affected. The world transforms of descendants must be recalculated to reflect the new position and orientation of the subtree's root node. It is possible that more than one change has been made at different locations in the scene graph. An implementation of a scene graph manager can attempt to maintain the minimum number of subtree root nodes that need to be updated. For example, if the local transforms are changed at nodes A and B, and if B is a descendant of A, the update of the subtree rooted at node A will automatically update the subtree rooted at B. It would be inefficient to first update the subtree at B, then update the subtree at A.

The updating is done in a recursive pass. Transforms are updated on the downward pass; bounding volumes are updated on the upward pass that is initiated as a return from the recursive calls. Note that the upward pass should not terminate at the node at which the initial update call was made. If the bounding volume of this node has changed as a result of changes in bounding volumes of the descendants, then the parent's bounding volume might also change. Thus, the upward pass must proceed

all the way to the root of the scene graph. If transforms are animated, the update pass is responsible for asking the controllers to make the necessary adjustments to the quantities they manage before the world transform is computed. Finally, if renderer state has changed, that information must be propagated to the leaf nodes (to support sorting as mentioned earlier). A single update call can be implemented to handle all changes in the scene graph, but since renderer state tends to change independently of geometry and transform changes, it might be desirable to have separate update passes.

The computation of model bounding volumes for geometric data was already discussed in Chapter 2. The main focus in the remainder of this section is on computing the parent's bounding volume from the child bounding volumes. The expense and algorithmic complexity depends on the type of volume used. It is possible to consider all child bounds simultaneously, but practice has shown that it is easier and faster to incrementally bound the children. For a node with three or more children, a bound is found for the first two children. That bound is increased in size to include the third child bound, and so on.

4.2.1 Merging Two Spheres

Source Code

Library
Containment

Filename
ContSphere

The algorithm described here computes the smallest sphere containing two spheres. Let the spheres S_i be $|\vec{X} - \vec{C}_i|^2 = r_i^2$ for $i = 0, 1$. Define $L = |\vec{C}_1 - \vec{C}_0|$ and unit-length vector $\vec{U} = (\vec{C}_1 - \vec{C}_0)/L$. The problem can be reduced to one dimension by projecting the spheres onto the line $\vec{C}_0 + t\vec{U}$. The projected intervals in terms of parameter t are $[-r_0, r_0]$ for S_0 and $[L - r_1, L + r_1]$ for S_1.

If $[-r_0, r_0] \subseteq [L - r_1, L + r_1]$, then $S_0 \subseteq S_1$ and the two spheres merge into S_1. The test for this case is $r_0 \le L + r_1$ and $L - r_1 \le -r_0$. A single test covers both conditions, $r_1 - r_0 \ge L$. To avoid the square root in computing L, compare instead $r_1 \ge r_0$ and $(r_1 - r_0)^2 \ge L^2$.

If $[L - r_1, L + r_1] \subseteq [-r_0, r_0]$, then $S_1 \subseteq S_0$ and the two spheres merge into S_0. The test for this case is $L + r_1 \le r_0$ and $-r_0 \le L - r_1$. A single test covers both conditions, $r_1 - r_0 \le -L$. Again to avoid the square root, compare instead $r_1 \le r_0$ and $(r_1 - r_0)^2 \ge L^2$.

Otherwise, the intervals either have partial overlap or are disjoint. The interval containing the two projected intervals is $[-r_0, L + r_1]$. The corresponding merged sphere whose projection is the containing interval has radius

$$r = \frac{L + r_1 + r_0}{2}.$$

The center t-value is $(L + r_1 - r_0)/2$ and corresponds to the point

$$\vec{C} = \vec{C}_0 + \frac{L + r_1 - r_0}{2}\vec{U} = \vec{C}_0 + \frac{L + r_1 - r_0}{2L}\left(\vec{C}_1 - \vec{C}_0\right).$$

The pseudocode is

```
Input:  Sphere(C0,r0) and Sphere(C1,r1)
centerDiff = C1 - C0;
radiusDiff = r1 - r0;
radiusDiffSqr = radiusDiff*radiusDiff;
Lsqr = centerDiff.SquaredLength();
if ( radiusDiffSqr >= LSqr )
{
    if ( radiusDiff >= 0.0f )
        return Sphere(C1,r1);
    else
        return Sphere(C0,r0);
}
else
{
    L = sqrt(Lsqr);
    t = (L+r1-r0)/(2*L);
    return Sphere(C0+t*centerDiff,(L+r1+r0)/2);
}
```

4.2.2 Merging Two Oriented Boxes

If two oriented boxes were built to contain two separate sets of data points, it is possible to build a single oriented bounding box that contains the union of the sets. That box might not contain the two original oriented boxes—something that is not desired in a hierarchical decomposition of an object. Moreover, the time it takes to build the single oriented box could be expensive.

Source Code

Library
Containment

Filename
ContBox

An alternative approach is to construct an oriented box from only the original boxes and that contains the original boxes. This can be done by interpolation of the box centers and axes, then growing the box to contain the originals. Let the original two boxes have centers $\vec{C}_i$ for $i = 0, 1$. Let the box axes be stored as columns of a rotation matrix R_i. Now represent the rotation matrices by unit quaternions q_i such that the dot product of the quaternions is nonnegative, $q_0 \cdot q_1 \geq 0$. The final box is assigned center $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$. The axes are obtained by interpolating the quaternions. The unit quaternion representing the final box is $q = (q_0 + q_1)/|q_0 + q_1|$, where the absolute value signs indicate length of the quaternion as a four-dimensional vector. The final box axes can be extracted from the quaternion using the methods described in Section 2.3. The extents of the final box are computed by projecting the vertices of the two original boxes onto the final box axes and computing the extreme values.

The pseudocode is

```
// Box has center, axis[3], extent[3]
Input:  Box box0, Box box1
Output:  Box box

// compute center
box.center = (box0.center + box1.center)/2;

// compute axes
Quaternion q0 = ConvertAxesToQuaternion(box0.axis);
Quaternion q1 = ConvertAxesToQuaternion(box1.axis);
Quaternion q = q0+q1;
Real length = Length(q);
q /= Length(q);
box.axis = ConvertQuaternionToAxes(q);

// compute extents
box.extent[0] = box.extent[1] = box.extent[2] = 0;
for each vertex V of box0 do
{
    Point3 delta = V - box.center;
    for (j = 0; j < 3; j++)
    {
        Real adot = |Dot(box0.axis[j],delta)|
        if ( adot > box.extent[j] )
            box.extent[j] = adot;
    }
}
for each vertex V of box1 do
{
    Point3 delta = V - box.center;
    for (j = 0; j < 3; j++)
    {
        Real adot = |Dot(box1.axis[j],delta)|
        if ( adot > box.extent[j] )
            box.extent[j] = adot;
    }
}
```

The function `ConvertAxesToQuaternion` stores the axes as columns of a rotation matrix, then uses the algorithm to convert a rotation matrix to a quaternion. The function `ConvertQuaternionToAxes` converts the quaternion to a rotation matrix, then extracts the axes as columns of the matrix.

4.2.3 Merging Two Capsules

Source Code

Library
Containment

Filename
ContCapsule

Two capsules may be merged into a single capsule with the following algorithm. If one capsule contains the other, just use the containing capsule. Otherwise, let the capsules have radii $r_i > 0$, end points $\vec{P}_i$, and directions $\vec{D}_i$ for $i = 0, 1$. The center points of the line segments are $\vec{C}_i = \vec{P}_i + \vec{D}_i/2$. Unit-length directions are $\vec{U}_i = \vec{D}/|\vec{D}|$.

The line L containing the final capsule axis is computed below. The origin of the line is the average of the centers of the original capsules, $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$. The direction vector of the line is obtained by averaging the unit direction vectors of the input capsules. Before doing so, the condition $\vec{U}_0 \cdot \vec{U}_1 \geq 0$ should be satisfied. If it is not, replace $\vec{U}_1$ by $-\vec{U}_1$. The direction vector for the line is $\vec{U} = (\vec{U}_0 + \vec{U}_1)/|\vec{U}_0 + \vec{U}_1|$.

The final capsule radius r must be chosen sufficiently large so that the final capsule contains the original capsules. It is enough to consider the spherical ends of the original capsules. The final radius is

$$r = \max\{\text{dist}(\vec{P}_0, L) + r_0, \text{dist}(\vec{P}_0 + \vec{D}_0) + r_0, \text{dist}(\vec{P}_1, L) + r_1, \text{dist}(\vec{P}_1 + \vec{D}_1, L) + r_1\}.$$

Observe that $r \geq r_i$ for $i = 0, 1$.

The final capsule direction $\vec{D}$ will be a scalar multiple of line direction $\vec{U}$. Let $\vec{E}_0$ and $\vec{E}_1$ be the end points for the final capsule, so $\vec{P} = \vec{E}_0$ and $\vec{D} = \vec{E}_1 - \vec{E}_0$. The end points must be chosen so that the final capsule contains the end spheres of the original capsules. Let the projections of $\vec{P}_0$, $\vec{P}_0 + \vec{D}_0$, $\vec{P}_1$, and $\vec{P}_1 + \vec{D}_1$ onto $\vec{C} + t\vec{U}$ have parameters τ_0, τ_1, τ_2, and τ_3, respectively. Let the corresponding capsule radii be denoted ρ_i for $0 \leq i \leq 3$. Let $\vec{E}_j = \vec{C} + T_j\vec{D}$ for $j = 0, 1$. The T_j are determined by "supporting" spheres that are selected from the end point spheres of the original capsules. If $\vec{Q}$ is the center of such a supporting sphere of radius ρ for end point $\vec{E}_1$, then T_1 is the smallest root of the equation $|\vec{C} + T\vec{U} - \vec{Q}| + \rho = r$. Since $r \geq \rho$, the equation can be written as a quadratic

$$T^2 + 2\vec{U} \cdot (\vec{C} - \vec{Q})T + |\vec{C} - \vec{Q}|^2 - (r - \rho)^2 = 0.$$

This equation must have only real-valued solutions. Similarly, if the $\vec{Q}$ is the center of the supporting sphere corresponding to end point $\vec{E}_0$, then T_0 is the largest root of the quadratic. The quadratics are solved for all four end points of the original capsules, and the appropriate minimum and maximum roots are chosen for the final T_0 and T_1.

4.2.4 Merging Two Lozenges

Two lozenges may be merged into a single lozenge that contains them with the following algorithm. Let the lozenges have radii $r_i > 0$, origins $\vec{P}_i$, and edges $\vec{E}_{ji}$ for $i = 0, 1$ and $j = 0, 1$. The center points of the rectangles of the lozenge are $\vec{C}_i = \vec{P}_i + (\vec{E}_{0i} + \vec{E}_{1i})/2$. Unit-length edge vectors are $\vec{U}_{ji} = \vec{E}_{ji}/|\vec{E}_{ji}|$. Unit-length normal vectors are $\vec{N}_i = \vec{U}_{0i} \times \vec{U}_{1i}$.

The center point of the final lozenge is the average of the centers of the original lozenges, $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$.

The edge vectors are obtained by averaging the coordinate frames of the original lozenges using a quaternion representation. Let q_i be the unit quaternion that represents the rotation matrix $[\vec{U}_{0i}\ \vec{U}_{1i}\ \vec{N}_i]$. If $q_0 \cdot q_1 < 0$, replace q_1 by $-q_1$. The final lozenge coordinate frame is extracted from the rotation matrix $[\vec{U}_0\ \vec{U}_1\ \vec{N}]$ corresponding to the unit quaternion $q = (q_0 + q_1)/|q_0 + q_1|$.

The problem now is to compute r sufficiently large so that the final lozenge contains the original lozenges. Project the original lozenges onto the line containing $\vec{P}$ and having direction $\vec{N}$. Each projection has extreme points determined by the corners of the projected rectangle and the radius of the original lozenge. The radius r of the final lozenge is selected to be the length of the smallest interval that contains all the extreme points of projection. Observe that $r \geq r_i$ is necessary.

Project the rectangle vertices of original lozenges onto the plane containing $\vec{P}$ and having normal $\vec{N}$. Compute the oriented bounding rectangle in that plane where the axes correspond to $\vec{U}_i$. This rectangle is associated with the final lozenge and produces the edges $\vec{E}_i = L_i\vec{U}_i$ for some scalars $L_i > 0$. The origin point for the final lozenge is $\vec{P} = \vec{C} - \vec{E}_0/2 - \vec{E}_1/2$.

Source Code

Library
Containment

Filename
ContLozenge

4.2.5 Merging Two Cylinders

To keep the merging algorithm simple, the original two cylinders are treated as capsules: their representations are converted to those for capsules, end points are $\vec{P}_i$, directions are $\vec{D}_i$, and radii are r_i. The capsule merging algorithm is applied to obtain the cylinder radius r. Rather than fitting a capsule to the points $\vec{P}_i \pm r_i\vec{U}$ and $\vec{P}_i + \vec{D}_i \pm r_i\vec{U}$, the points are projected onto the line $\vec{P} + t\vec{D}$, where $\vec{P}$ is suitably chosen from one of the fitting algorithms. The smallest interval containing the projected points determines cylinder height h.

Source Code

Library
Containment

Filename
ContCylinder

4.2.6 Merging Two Ellipsoids

Computing a bounding ellipsoid for two other ellipsoids is done in a way similar to that of oriented boxes. The ellipsoid centers are averaged, and the quaternions representing the ellipsoid axes are averaged and then the average is normalized. The original ellipsoids are projected onto the newly constructed axes. On each axis, the smallest interval of the form $[-\sigma, \sigma]$ is computed to contain the intervals of projection. The σ-values determine the minor axis lengths for the final ellipsoid.

Source Code

Library
Containment

Filename
ContEllipsoid

4.2.7 Algorithm for Scene Graph Updating

The pseudocode for updating the spatial information in a scene graph is given below. Three abstract classifications are used: `Spatial`, `Geometry`, and `Node`. In an object-

Source Code

Library
Engine

Filename
Spatial
Geometry
Node

oriented implementation, the last two classes are both derived from `Spatial`. The `Spatial` class manages a link to a parent, local transforms, and a world transform. It represents leaf nodes in a tree. The `Node` class manages links to children. It represents internal nodes in the tree. The `Geometry` class represents leaf nodes that contain geometric data. It manages a model bounding volume.

The entry point into the update system for geometric state (GS) is

```
void Spatial::UpdateGS (float time, bool initiator)
{
    UpdateWorldData(time);
    UpdateWorldBound();
    if ( initiator )
        PropagateBoundToRoot();
}
```

The input parameter to the call is set to `true` by the node at which the update is initiated. This allows the calling node to propagate the world bounding volume update to the root of the scene graph.

The function `UpdateWorldData` is virtual and controls the downward pass that computes world transforms and updates time-varying quantities:

```
virtual void Spatial::UpdateWorldData (float time)
{
    // update dynamically changing render state
    for each render state controller rcontroller do
        rcontroller.Update(time);

    // update local transforms if managed by controllers
    for each transform controller tcontroller do
        tcontroller.Update(time);

    // Compute product of parent's world transform with this object's
    // local transform.  If no parent exists, the child's world
    // transform is just its local transform.

    if ( world transform not computed by a transform controller )
    {
        if ( parent exists )
        {
            worldScale = parent.worldScale*localScale;
            worldRotate = parent.worldRotate*localRotate;
            worldTranslate = parent.worldTranslate +
                parent.worldScale*(parent.worldRotate*localTranslate);
        }
```

```
        else
        {
            // node is the root of the scene graph
            worldScale = localScale;
            worldRotate = localRotate;
            worldTranslate = localTranslate;
        }
    }
}
```

The function `UpdateWorldBound` is also virtual and controls the upward pass and allows each node object to update its world bounding volume. Base class `Spatial` has no knowledge of geometric data and in particular does not manage a model bounding sphere, so the function is pure virtual and must be implemented both by `Geometry`, which knows how to transform a model bounding volume to a world bounding volume, and by `Node`, which knows how to merge world bounding volumes of its children.

Finally, the propagation of world bounding volumes is not virtual and is a simple recursive call:

```
void Spatial::PropagateBoundToRoot ()
{
    if ( parent exists )
    {
        parent.UpdateWorldBound();
        parent.PropagateBoundToRoot();
    }
}
```

The derived classes override the virtual functions. Class `Geometry` has nothing more to say about updating world data, but it must update the world bound,

```
virtual void Geometry::UpdateWorldBound ()
{
    worldBound = modelBound.TransformBy(worldRotate,
      worldTranslate,worldScale);
}
```

The model bound is assumed to be correct. If model data is changed, the application is required to update the model bound.

Class `Node` updates are as shown:

```
virtual void Node::UpdateWorldData (float time)
{
    Spatial::UpdateWorldData(time);
```

```
    for each child do
        child.UpdateGS(false); // child not initiator of
                               // original UpdateGS call
}

virtual void Node::UpdateWorldBound ()
{
    worldBound = firstChild.GetWorldBound();
    for each additional child do
        worldBound = Merge(worldBound,child.worldBound);
}
```

The downward pass is controlled by `UpdateWorldData`. The node first updates its world transforms by a call to the base class update of world transforms. The children of the node are each given a chance to update themselves, thus yielding a recursive chain of calls involving `UpdateGS` and `UpdateWorldData`. The update of world bounds is done incrementally. The world bound is set to the first child's world bound. As each remaining child is visited, the current world bound and the child world bound are merged into a single bound that contains both. Although this approach usually does not produce the tightest bound, it is much faster than methods that do attempt the tightest bound. For example, if bounding spheres are used, it is possible to compute the parent world bound as the minimum volume sphere containing any geometric data of the descendants. Such a computation is expensive and will severely affect the frame rate of the application. The trade-off is to obtain a reasonable world bounding volume for the parent that is inexpensive to compute.

Updating the set of current renderer states at the leaf nodes is also a recursive system just as `UpdateGS` is. Class `Geometry` maintains a set of such states; call that member `stateSet`. Each state can be attached to or detached from an object of this class. A state object itself has information that can be modified at run time. If the information is changed, then an update must occur starting at that node. The global renderer state set is maintained by the renderer, so any changes to renderer state by the objects must be communicated to the renderer. Class `Spatial` provides the virtual function foundation for the renderer state (RS) update:

```
void Spatial::UpdateRS (RenderState parentState)
{
    // update render states
    if ( parentState exists )
    {
        // parentState must remain intact to restore state after
        // recursion
        currentState = parentState;
        modify currentState with thisState;
    }
```

```
    else
    {
        // this object is initiator of UpdateRS, use default
        // renderer states
        currentState = defaultRenderState;
        PropagateStateFromRoot(currentState);
    }

    UpdateRenderState(currentState);
}
```

The initial call to `UpdateRS` is typically applied to a node in the tree that is not the root node. Any renderer state from predecessors of the initiating node must be accumulated before the downward recursive pass. The function `PropagateStateFromRoot` does this work:

```
    void Spatial::PropagateStateFromRoot (RenderState
                                          currentState)
    {
        // traverse to root to allow downward state propagation
        if ( parent exists )
            parent.PropagateStateFromRoot(currentState);

        // update parent state by current state
        modify currentState with thisState;
}
```

The call `UpdateRenderState` is pure virtual. Class `Geometry` implements this to update its renderer state at leaf nodes. Class `Node` implements this to perform the recursive traversal of the call on its children.

```
    void Geometry::UpdateRenderState (RenderState currentState)
    {
        modify thisState with currentState;
    }

    void Node::UpdateRenderState (RenderState currentState)
    {
        for each child do
            child.UpdateRS(currentState);
    }
}
```

Notice that `UpdateRS` and `UpdateRenderState` form a recursive chain just as `UpdateGS` and `UpdateWorldData` form a recursive chain.

4.3 Rendering a Scene Graph

The renderer manages a camera whose job it is to define the *view frustum,* the portion of the world to be viewed. The process of rendering the scene graph in the frustum at a given instant is typically referred to as the *camera click*. This process involves a traversal of the scene graph, and the graph is assumed to be current (as established by the necessary `UpdateGS()` and `UpdateRS()` calls at the relevant nodes).

Scene graph traversal includes object level culling as described earlier. If the world bounding volume for a node is outside the view frustum, then the subtree rooted at that node need not be traversed. If a subtree is not culled, then the traversal is recursive. The renderer states are collected during traversal until a leaf node of the scene graph is reached. At this point the renderer has all the state information it requires to be able to properly draw the geometry represented by the leaf node. The leaf node has the responsibility of providing the renderer with its geometric data such as vertices, triangle connectivity information, triangle normals (for back face culling), and surface attributes including vertex normals, colors, and texture coordinates.

Before the actual rendering of the leaf node object, it is useful to allow the object to perform any preparations that are necessary for proper display. For example, culling is based on world bounding volumes. The classes derived from `Geometry` have the liberty of keeping current the world bounding sphere via the `UpdateWorldBound` call. If an object is to be culled, then computing any expensive world data in the call to `UpdateWorldData` is wasteful. Instead, the `Geometry` classes could provide a Boolean flag indicating whether or not the world data is current. The call to `UpdateWorldData` updates world transforms, but additionally sets only the Boolean flag indicating the world data is not current. A prerendering function called *after* it is determined that an object is not to be culled can test the Boolean flag, find out the world data is not current, make the data current, then set the flag to indicate the data is current.

Another use of a prerendering function involves dynamic tessellation of an object. Chapter 10 discusses objects represented by a triangular mesh whose triangles are increased or reduced based on a continuous level-of-detail algorithm involving a preprocessed set of incremental mesh changes. The prerendering function can select the appropriate level of detail based on the current camera and view frustum. Chapter 8 discusses objects represented by curved surfaces. The prerendering function can dynamically tessellate the surfaces to the appropriate level of detail.

The complement of a prerendering function is a postrendering function that gives the object a chance to do any cleanup associated with prerendering and actual rendering.

4.3.1 Culling by Spheres

The test for intersection of bounding volume with view frustum is performed in world space since the world bounding information is kept current by the object and the world view frustum information is kept current by the camera. Let the world

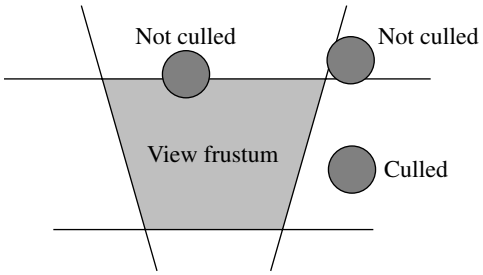

Figure 4.2 Examples of culled and unculled objects.

Source Code

Library
Intersection

Filename
IntrPlnSphr

bounding sphere have center $\vec{C}$ and radius r. Let a view frustum plane be specified by $\vec{N} \cdot \vec{X} = d$, where $\vec{N}$ is a unit-length vector that points to the interior of the frustum. The bounding sphere does not intersect the frustum when the distance from $\vec{C}$ to the plane is larger than the sphere radius. An object is completely culled if its bounding sphere satisfies

$$\vec{N} \cdot \vec{C} - d < -r \tag{4.8}$$

for one of the frustum planes. The left-hand side of the inequality is the signed distance from $\vec{C}$ to the plane. The right-hand side is negative and indicates that to be culled, $\vec{C}$ must be on the outside of the frustum plane and must be at least the sphere radius units away from the plane. The test requires 3 multiplications and 3 additions. The pseudocode is

```
bool CullSpherePlane (Sphere sphere, Plane plane)
{
    return Dot(plane.N,sphere.C) - plane.d < -sphere.r;
}
```

It is possible for a bounding sphere to be outside the frustum even if all six culling tests fail. Figure 4.2 shows examples of an object that is culled by the tests. It also shows examples of objects that are not culled, one object whose bounding sphere intersects the frustum and one object whose bounding sphere does not intersect the frustum. In either case, the object must be further processed in the clipping pipeline. Alternatively, the exact distance from bounding sphere to frustum can be computed at greater expense than the distances from sphere to planes.

Better-fitting bounding volumes can lead to rejection of an object when the bounding sphere does not, thereby leading to savings in CPU cycles. However, the

application must keep the bounding volume current as the object moves about the world. For each change in a rigid object's orientation, the bounding volume must be rotated accordingly. This leads to a trade-off between more time to update bounding volume and less time to process objects because they are more accurately culled.

The following sections describe the culling algorithms for oriented boxes, capsules, lozenges, cylinders, and ellipsoids. In each section the frustum plane is $\vec{N} \cdot \vec{X} = d$ with unit-length normal pointing to frustum interior.

4.3.2 Culling by Oriented Boxes

An oriented bounding box is outside the frustum plane if all its vertices are outside the plane. The obvious algorithm of testing if all eight vertices are on the "negative side" of the plane requires eight comparisons of the form $\vec{N} \cdot \vec{V} < d$. The vertices are of the form

Source Code

Library
Intersection

Filename
IntrPlnBox3

$$\vec{V} = \vec{C} + \sigma_0 a_0 \vec{A}_0 + \sigma_1 a_1 \vec{A}_1 + \sigma_2 a_2 \vec{A}_2,$$

where $|\sigma_i| = 1$ for all i (eight possible choices, two for each σ_i). Each test requires computing signed distances

$$\vec{N} \cdot \vec{V} - d = (\vec{N} \cdot \vec{C} - d) + \sigma_0 a_0 \vec{N} \cdot \vec{A}_0 + \sigma_1 a_1 \vec{N} \cdot \vec{A}_1 + \sigma_2 a_2 \vec{N} \cdot \vec{A}_2.$$

The 4 dot products are computed once, each dot product using 3 multiplications and 2 additions. Each test requires an additional 3 multiplications and 4 additions (the multiplications by σ_i are not counted). The eight tests therefore require 36 multiplications and 40 additions.

A faster test is to project the box and plane onto the line $\vec{C} + s\vec{N}$. The symmetry provided by the box definition yields an interval of projection $[\vec{C} - r\vec{N}, \vec{C} + r\vec{N}]$. The interval is centered at $\vec{C}$ and has radius

$$r = a_0 |\vec{N} \cdot \vec{A}_0| + a_1 |\vec{N} \cdot \vec{A}_1| + a_2 |\vec{N} \cdot \vec{A}_2|.$$

The frustum plane projects to a single point

$$\vec{P} = \vec{C} + (d - \vec{N} \cdot \vec{C})\vec{N}.$$

The box is outside the plane as long as the projected interval is outside, in which case $\vec{N} \cdot \vec{C} - d < -r$. The test is identical to that of sphere-versus-plane, except that r is known for the sphere but must be calculated for each test of an oriented bounding box. The test requires 4 dot products, 3 multiplications, and 3 additions for a total operation count of 15 multiplications and 11 additions. The pseudocode is

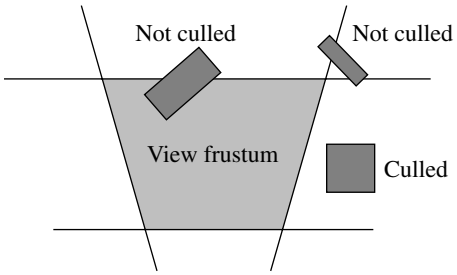

Figure 4.3 Examples of culled and unculled objects.

```
bool CullBoxPlane (Box box, Plane plane)
{
    r = box.a0*|Dot(plane.N,box.A0)| +
        box.a1*|Dot(plane.N,box.A1)| +
        box.a2*|Dot(plane.N,box.A2)|;

    return Dot(plane.N,box.C) - plane.d < -r;
}
```

As with the sphere, it is possible for an oriented bounding box not to be culled when tested against each frustum plane one at a time, even though the box is outside the view frustum. Figure 4.3 illustrates such a situation.

4.3.3 Culling by Capsules

Source Code

Library
Intersection

Filename
IntrPlnCap

A capsule consists of a radius $r > 0$ and a parameterized line segment $\vec{P} + t\vec{D}$, where $\vec{D} \neq \vec{0}$ and $t \in [0, 1]$. The signed distances from plane to end points are $\delta_0 = \vec{N} \cdot \vec{P} - d$ and $\delta_1 = \vec{N} \cdot (\vec{P} + \vec{D}) - d$. If either $\delta_0 \geq 0$ or $\delta_1 \geq 0$, then the capsule is not culled since it is either intersecting the frustum plane or on the frustum side of the plane. Otherwise, both signed distances are negative. If $\vec{N} \cdot \vec{D} \leq 0$, then end point $\vec{P}$ is closer in signed distance to the frustum plane than is the other end point $\vec{P} + \vec{D}$. The distance between $\vec{P}$ and the plane is computed and compared to the capsule radius. If $\vec{N} \cdot \vec{P} - d \leq -r$, then the capsule is outside the frustum plane and it is culled; otherwise it is not culled. If $\vec{N} \cdot \vec{D} > 0$, then $\vec{P} + \vec{D}$ is closer in signed distance to the frustum plane than is $\vec{P}$. If $\vec{N} \cdot (\vec{P} + \vec{D}) - d \leq -r$, then the capsule is culled; otherwise it is not culled. The pseudocode for the culling algorithm is given below. The Boolean result is `true` if and only if the capsule is culled.

```
bool CullCapsulePlane (Capsule capsule, Plane plane)
{
    sd0 = Dot(plane.N,capsule.P) - plane.d;
    if ( sd0 < 0 )
    {
        sd1 = sd0 + Dot(plane.N,capsule.D);
        if ( sd1 < 0 )
        {
            if ( sd0 <= sd1 )
            {
                // P0 closest to plane
                return sd0 <= -capsule.r;
            }
            else
            {
                // P1 closest to plane
                return sd1 <= -capsule.r;
            }
        }
    }

    return false;
}
```

4.3.4 Culling by Lozenges

SOURCE CODE

LIBRARY
Intersection

FILENAME
IntrPlnLoz

A lozenge consists of a radius $r > 0$ and a parameterized rectangle $\vec{P} + s\vec{E}_0 + t\vec{E}_1$, where $\vec{E}_0 \neq \vec{0}$, $\vec{E}_1 \neq \vec{0}$, $\vec{E}_0 \cdot \vec{E}_1 = 0$, and $(s, t) \in [0, 1]^2$. The four rectangle corners are $\vec{P}_{00} = \vec{P}$, $\vec{P}_{10} = \vec{P} + \vec{E}_0$, $\vec{P}_{01} = \vec{P} + \vec{E}_1$, and $\vec{P}_{11} = \vec{P} + \vec{E}_0 + \vec{E}_1$. The signed distances are $\delta_{ij} = \vec{N} \cdot \vec{P}_{ij} - d$. If any of the signed distances are nonnegative, then the lozenge either intersects the plane or is on the frustum side of the plane and it is not culled. Otherwise, all four signed distances are negative. The rectangle corner closest to the frustum plane is determined, and its distance to the plane is compared to the lozenge radius to determine if there is an intersection. The pseudocode for the culling algorithm is

```
bool CullLozengePlane (Lozenge lozenge, Plane P)
{
    sd00 = Dot(plane.N,lozenge.P) - plane.d;
    if ( sd00 < 0 )
    {
        dotNE0 = Dot(plane.N,lozenge.E0);
        sd10 = sd00 + dotNE0;
```

```
            if ( sd10 < 0 )
            {
                dotNE1 = Dot(plane.N,lozenge.E1);
                sd01 = sd00 + dotNE1;
                if ( sd01 < 0 )
                {
                    sd11 = sd10 + dotNE1;
                    if ( sd11 < 0 )
                    {
                        // all rectangle corners on negative side
                        // of plane
                        if ( sd00 <= sd10 )
                        {
                            if ( sd00 <= sd01 )
                            {
                                // P00 closest to plane
                                return sd00 <= -lozenge.r;
                            }
                            else
                            {
                                // P01 closest to plane
                                return sd01 <= -lozenge.r;
                            }
                        }
                        else
                        {
                            if ( sd10 <= sd11 )
                            {
                                // P10 closest to plane
                                return sd10 <= -lozenge.r;
                            }
                            else
                            {
                                // P11 closest to plane
                                return sd11 <= -lozenge.r;
                            }
                        }
                    }
                }
            }

            return false;
        }
```

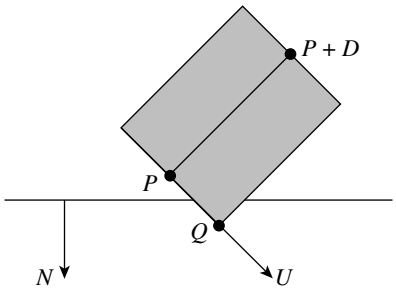

Figure 4.4 Projection of cylinder and frustum plane, no-cull case.

4.3.5 Culling by Cylinders

Source Code

Library
Intersection

Filename
IntrPlnCyln

A cylinder consists of a radius $r > 0$, a height $h \in [0, \infty]$, and a parameterized line segment $\vec{C} + t\vec{W}$, where $|\vec{W}| = 1$ and $t \in [-h/2, h/2]$. Figure 4.4 shows a typical no-cull situation. Let the plane be $\vec{N} \cdot \vec{X} = d$, where $|\vec{N}| = 1$. Let $\vec{U}$, $\vec{V}$, and $\vec{W}$ form an orthonormal set of vectors. Any cylinder point $\vec{X}$ can be written as $\vec{X} = \vec{C} + y_0\vec{U} + y_1\vec{V} + y_2\vec{W}$, where $y_0^2 + y_1^2 = r^2$ and $|y_2| <= h/2$. Let $y_0 = r\cos(A)$ and $y_1 = r\sin(A)$. Substitute $\vec{X}$ in the plane equation to get

$$-(\vec{N} \cdot \vec{W})y_2 = (\vec{N} \cdot \vec{C} - d) + (\vec{N} \cdot \vec{U})r\cos(A) + (\vec{N} \cdot \vec{V})r\sin(A).$$

If $\vec{N} \cdot \vec{W} = 0$, then the plane is parallel to the axis of the cylinder. The two intersect if and only if the distance from $\vec{C}$ to the plane satisfies

$$|\vec{N} \cdot \vec{C} - d| \le r.$$

In this situation the cylinder is culled when $\vec{N} \cdot \vec{C} - d \le -r$.

If $\vec{N} \cdot \vec{W} \neq 0$, then y_2 is a function of A. The minimum and maximum values can be found by the methods of calculus. The extreme values are

$$\frac{d - \vec{N} \cdot \vec{C} \pm \sqrt{1 - (\vec{N} \cdot \vec{W})^2}}{\vec{N} \cdot \vec{W}}.$$

The plane and cylinder intersect if and only if

$$\min(y_2) \le h/2 \quad \text{and} \quad \max(y_2) \ge -h/2.$$

In this situation the cylinder is culled when the previous tests show no intersection and $\vec{N} \cdot \vec{C} - d \leq -r$. The pseudocode is

```
bool CullCylinderPlane (Cylinder cylinder, Plane plane)
{
    sd0 = Dot(plane.N,cylinder.P) - plane.d;
    if ( sd0 < 0 )
    {
        dotND = Dot(plane.N,cylinder.D)
        sd1 = sd0 + dotND;
        if ( sd1 < 0 )
        {
            dotDD = Dot(cylinder.D,cylinder.D);
            r2 = cylinder.r*cylinder.r;
            if ( sd0 <= sd1 )
            {
                // P0 closest to plane
                return dotDD*sd0*sd0 >= r2*(dotDD-dotND*dotND);
            }
            else
            {
                // P1 closest to plane
                return dotDD*sd1*sd1 >= r2*(dotDD-dotND*dotND);
            }
        }
    }

    return false;
}
```

The quantities $\vec{D} \cdot \vec{D}$ and r^2 can be precomputed and stored by the cylinder as a way of reducing execution time for the intersection test.

4.3.6 Culling by Ellipsoids

Source Code

Library
Intersection

Filename
IntrPlnElp3

An ellipsoid is represented by the quadratic equation $Q(\vec{X}) = (\vec{X} - \vec{C})^{\mathrm{T}} M (\vec{X} - \vec{C}) = 1$, where $\vec{C}$ is the center of the ellipsoid, where M is a positive definite matrix, and where $\vec{X}$ is any point on the ellipsoid. An ellipsoid is outside a frustum plane whenever the projection of the ellipsoid onto the line $\vec{C} + s\vec{N}$ is outside the frustum plane. The projected interval is $[-r, r]$. Figure 4.5 shows a typical no-cull situation. The ellipsoid is culled whenever

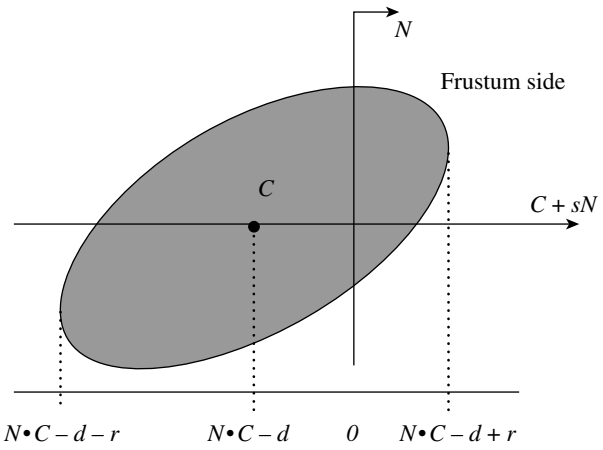

Figure 4.5 Projection of ellipsoid and frustum plane, no-cull case.

$\vec{N} \cdot \vec{C} - d \le -r.$

The construction of r is as follows. The points $\vec{X}$ that project to the end points of the interval must occur where the normals to the ellipsoid are parallel to $\vec{N}$. The gradient of $Q(\vec{X})$ is a normal direction for the point, $\vec{\nabla} Q = 2M(\vec{X} - \vec{C})$. Thus, $\vec{X}$ must be a solution to $M(\vec{X} - \vec{C}) = \lambda \vec{N}$ for some scalar λ. Inverting M and multiplying yields $\vec{X} - \vec{C} = \lambda M^{-1}\vec{N}$. Replacing this in the quadratic equation yields $1 = \lambda^2 (M^{-1}\vec{N})^{\mathrm{T}} M (M^{-1}\vec{N}) = \lambda^2 \vec{N}^{\mathrm{T}} M^{-1} \vec{N}$. Finally, $r = \vec{N} \cdot (\vec{X} - \vec{C}) = \lambda \vec{N}^{\mathrm{T}} M^{-1} \vec{N}$, so $r = \sqrt{\vec{N}^{\mathrm{T}} M^{-1} \vec{N}}$. The pseudocode is

```
bool CullEllipsoidPlane (Ellipsoid ellipsoid, Plane plane)
{
    sd0 = Dot(plane.N,ellipsoid.C) - plane.d;
    if ( sd0 < 0 )
    {
        r2 = Dot(plane.N,ellipsoid.Minverse*plane.N);
        return sd0*sd0 >= r2;
    }

    return false;
}
```

4.3.7 Algorithm for Scene Graph Rendering

An abstract class `Renderer` has a method that is the entry point for drawing a scene graph:

Source Code

Library
Engine

Filename
Renderer
Spatial
Geometry
Node
TriMesh

```
void Renderer::Draw (Spatial scene)
{
    scene.OnDraw(thisRenderer);
}
```

Its sole job is to start the scene graph traversal and pass the renderer for camera access and for accumulating render state. The method is virtual so that any derived class renderer can perform any setup before, and any cleanup after, the scene graph is drawn.

The class `Spatial` implements

```
void Spatial::OnDraw (Renderer renderer)
{
    if ( forceCulling )
        return;

    savePlaneState = renderer.planeState;

    if ( !renderer.Cull(worldBound) )
        Draw(renderer);

    renderer.planeState = savePlaneState;
}
```

The class `Spatial` provides a Boolean flag to allow the application to force culling of an object. If the object is not forced to be culled, then comparison of the world bounding volume to the camera frustum planes is done next. As mentioned in Section 3.4, if the bounding volumes are properly nested, once a bounding volume is inside a frustum plane there is no need to test bounding volumes of descendants against that plane. In this case the plane is said to be *inactive*. The renderer keeps track of which planes are active and inactive (the plane state). The current object must save the current plane state since the state might change during the recursive pass and the old state must be restored.

The member function `Draw` of class `Spatial` is also a pure virtual function. Class `Geometry` manages the leaf node renderer state and uses the `Draw` function to tell the renderer about the state it should use for drawing that leaf node. Class `Node` again provides for the recursive propagation to its children.

```
void Geometry::Draw (Renderer renderer)
{
    renderer.SetState(thisState);
}

void Node::Draw (Renderer renderer)
{
    for each child do
        child.OnDraw(renderer);
}
```

Notice the pattern of recursive chains provided by classes `Spatial` and `Node`. In this case `Draw` and `OnDraw` form the recursive chain.

Finally, for a specific class derived from `Geometry` that has actual data, the renderer must implement how to draw that data. For example, if `TriMesh` is derived from `Geometry` and manages a triangle mesh with vertices, normals, colors, and texture coordinates, the class must implement the virtual function as

```
void TriMesh::Draw (Renderer renderer)
{
    Geometry::Draw(renderer);
    renderer.Draw(this);
}
```

The call to the base class `Draw` tells the renderer to use the current rendering state at the leaf node. The next call allows the renderer to do its specific work with the triangle mesh. The `Draw` call in the renderer is a pure virtual function. If class `SoftRender` is derived from `Renderer` and represents software rendering, then the entire geometric pipeline of transformation, clipping, projection, and rasterizing is encapsulated in `Draw` for `SoftRender`. On the other hand, if class `HardRender` is derived from `Renderer` and represents a hardware-accelerated renderer, then `Draw` probably does very little work and can feed the hardware card directly.