Cosmo 3D™ Programmer's Guide

Document Number 007-3445-002

CONTRIBUTORS

Written by George Eckel
Illustrated by Dany Galgani and Martha Levine
Production by Carlos Miqueo
Engineering contributions by Brian Cabral, John Rohlf, Brad Grantham, Chris Tanner, Rich Silba, Tonia Spyridi, Michael Jones, Trina Roy, Chris Walker
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Cosmo 3D™ Programmer's Guide
Document Number 007-3445-002

Contents at a Glance

Contents

List of Figures

List of Tables

About This Guide

Cosmo 3D is a new toolkit that brings 3D graphics programming to desktop applications. Cosmo 3D is a scene graph API; its concepts are new, but similar to concepts developed in Open Inventor, Performer, and OpenGL.

This guide shows you how to develop Cosmo 3D applications. Included are descriptions of Cosmo 3D applications that you can run on your workstation, as well as code examples that you can use as a guide when developing your Cosmo 3D applications.

This guide presents the developer's view of the Cosmo 3D's C++ library with C++ examples.

What This Guide Contains

This guide presents information about Cosmo 3D in a task-oriented manner: the topics in this guide are arranged to coincide with the order in which you need to refer to them while writing a Cosmo 3D application. To illustrate the use of Cosmo 3D, code examples are sprinkled throughout the guide. Additional sample source code is provided in the */usr/share/optimizer/cosmo1.1/cosmo/test/C++* directory.

Brief descriptions of the chapters in this guide follow:

- Chapter 1, "Getting Started with Cosmo 3D," provides an overview of Cosmo 3D, introduces some of its most basic classes, and lists the steps involved in creating a typical application.
- Chapter 2, "Creating Geometries," discusses large, ready-made geometries, such as **csSphere** and **csCube** objects, and explains how to use the **csGeoSet**-derived classes provided by Cosmo 3D and how to create your own **csGeoSet**-derived classes.

- Chapter 3, "Specifying the Appearance of Geometries," describes the appearance fields in **csContext** and **csAppearance**.
- Chapter 4, "Scene Graph Nodes," describes nodes and node types.
- Chapter 5, "Building a Scene Graph," describes how to build and edit a scene graph.
- Chapter 6, "Placing Shapes in a Scene," describes how to place shapes in scenes.
- Chapter 7, "Traversing the Scene Graph," describes how an action traverses a scene graph and a description of the actions available in Cosmo 3D.
- Chapter 8, "Lighting and Fog," describes how to use lights, change the shadow modeling, and change the screen to one color. It also discusses fog, a new feature in Cosmo 3D 1.1.
- Chapter 9, "Viewing the Scene," describes how to set up the viewport and how to use cameras to view a scene.
- Chapter 10, "Scene Graph Engines," describes **csEngine** and the multiple subclasses derived from it.
- Chapter 11, "Sensors," explains how to implement sensors. Sensors are used to detect time passing and ointer device events.
- Chapter 12, "User Interface Mechanisms," discusses how to implement user interaction using X window code, **csWindow**, and selection mechanisms.
- Chapter 13, "Multiprocessing," describes how to implement multiprocessing.
- Chapter 14, "Optimizing Rendering," describes the Cosmo 3D nodes and programming techniques that can help optimize your application's performance.
- Chapter 15, "Adding Sounds To Virtual Worlds," describes how to set and play sound using Cosmo 3D.
- Appendix A, "Cosmo Basic Types," discusses all of the basic types that are used in other Cosmo 3D classes.
- Appendix B, "Cosmo 3D Sample Application," lists a complete sample application and explains its components.
- Appendix C, "Cosmo 3D Class Hierarchy," shows the class hierarcy in Cosmo 3D.

These chapters and appendices are followed by an index.

Related Reading

Reference pages for Cosmo 3D are obtained by pointing your web browser at:

- For IRIX: */usr/share/Optimizer/doc/developer*
- For Windows: *<inst_dir>/doc/developer*

Where *inst_dir* is the directory where Optimizer was installed. The default installation location is *<system_drive>:/Progral Files/Silicon Graphics/Optimizer.*

Who Should Read This Guide

This guide is written for developers of OpenGL Optimizer applications. Developers use Cosmo 3D scene graph nodes and actions to develop OpenGL Optimizer applications.

What You Should Know Before Reading This Guide

This guide is written with the assumption that the reader is experienced with C++.

Suggestions for Further Reading

For information on Open Inventor, see the following:

- Wernecke, Josie, *The Inventor Mentor*. Reading, Mass.:Addison Wesley 1994
- Wernecke, Josie, *The Inventor Toolmaker*. Reading, Mass.:Addison Wesley 1994
- Open Inventor Architecture Group, *Open Inventor C++ Reference Manual*. Reading, Mass.:Addison Wesley 1994
- OpenGL Architecture Review Board, M. Woo, J. Neider, and Tom Davis, *OpenGL Programming Guide*, Second Edition, 1997. (Also known as “the Red book.”)

For information on OpenGL Optimizer; see the following SGI manual:

OpenGL Optimizer Programmer’s Guide: An Open API for Large-Model Visualization (document number 007-2852-002).

Style Conventions

These style conventions are used in this guide:

- **Bold**—Functions, class names, node names, data members, and data types
- *Italics*—Variables, filenames, spatial dimensions, and commands
- Regular—Program names and enumerated types

Code examples are set off from the text in a fixed-space font.

Chapter 1

Getting Started with Cosmo 3D

Cosmo 3D is a scene graph API that brings 3D graphics programming to desktop applications. Cosmo 3D speeds up and facilitates the process of creating complex graphics applications. It allows applications to use a higher-level interface than the lower-level OpenGL language that it is based on. Developers interact with C++ objects that are arranged in an object hierarchy.

With its scene graph architecture and features such as culling, level of detail (LOD), 2D texture mapping, and audio, Cosmo 3D enables you to develop complex graphic applications, for example, professional character animations and gaming applications.

After creating a scene graph using Cosmo 3D objects, developers can use the OpenGL Optimizer API to improve performance. See the manual *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization* for more information.

This chapter gives an overview of the base classes of a Cosmo 3D scene graph. Understanding how each class contributes to the scene graph is essential for making optimal use of the API. These are the sections in this chapter:

- "Understanding a Cosmo 3D Scene Graph" on page 2.
- "Scene Graph Base Classes" on page 2.
- "Scene Graph Construction Classes" on page 7.
- "Classes That Determine How Things Are Drawn" on page 12.
- "Classes defining Geometric Objects" on page 13.
- "Steps for Creating and Displaying a Simple Scene Graph" on page 13.

Understanding a Cosmo 3D Scene Graph

A scene graph is a directed acyclical graph of nodes that embodies the semantics of *what* is to be drawn, but not *how* it is to be drawn. Developers interacting with a scene graph are interested in achieving a result, usually seeing a model on screen and manipulating it. They leave it up to Cosmo 3D to achieve this result in the most efficient way.

A Cosmo 3D scene graph consists of objects that inherit appropriate methods and fields from the Cosmo 3D classes. Conceptually, there are four kinds of classes:

- Base classes—**csObject**, **csField**, **csContainer**, and **csNode**. These classes are never instantiated directly. Instead, applications create subclasses that inherit certain functionality from the base classes. Base classes are discussed in this chapter.
- Scene graph construction classes—**csGroup**, **csShape**, **csGeometry**, and **csAppearance** determine appearance in a general way.
- Specific appearance classes—**csContext**, **csDrawTraversal**, **csEnvironment** and some of their subclasses determine how things are drawn, for example, whether lights or fog are applied.
- Geometry classes, such as **csSphere** or **csCylinder**, are the building blocks of the model itself.

This manual starts by discussing the different kinds of classes. It then briefly lists the steps required to create a simple sample program. The sample program itself is listed in Appendix B, "Cosmo 3D Sample Application."

Scene Graph Base Classes

This section discusses the following abstract, base classes that provide the functionality that is necessary to implement a scene graph:

- "The csObject Class"
- "The csContainer Class"
- "The csField Class"
- "The csNode Class"

The csObject Class

The **csObject** class is the base class for all objects in a scene; where an object is an entity that you can place in the scene graph. A **csObject** provides reference counting and runtime typing for all its children.

Reference Counting

Many kinds of data objects in Cosmo 3D can be placed in a hierarchical scene graph. Using instancing, an object can be referenced multiple times. Scene graphs can become quite complex, which can cause problems if you're not careful. Deleting objects can be a particularly dangerous operation, for example, if you delete an object that another object still references.

Within each **csObject** is a counter that keeps track of the number of objects referencing a particular instance. Reference counting provides a bookkeeping mechanism that makes object deletion safe: an object should never be deleted if its reference count is greater than zero. In general, you should only unreference an object in case it is referenced by another object.

It is just as important, however, not to unreference an object that has not been referenced. Because the reference count is an unsigned integer, unreferencing an object that has not been referenced decrements the reference count from 0 to a large positive number and it will never be deleted.

Each **csObject** is created with a reference count of 0. It is important to reference an object when it is created to make sure that someone else does not delete it when they unreference it.

When object A is attached to object B, the reference count of A is incremented. Additionally, if A replaces a previously referenced object C, the reference count of C is decremented.

Example 1-1 demonstrates how reference counts are incremented and decremented.

Example 1-1 Objects and Reference Counts

```
csAppearance *appearanceA, *appearanceC;
csGeoSet *gset;
csShape *shape;

shape->setGeometry(0, gset);
/* Attach appearanceC to gset. Reference count of appearanceC
 * is incremented. */
shape->setAppearance(appearanceC);

/* Attach appearanceA to gset, replacing appearanceC. Reference
 * count of appearanceC is decremented and that of appearanceA
 * is incremented. */
shape->setAppearance(appearanceA);
```

When the reference count of an existing **csObject** becomes 0, the object is assumed not to be referenced by any other object and is deleted. An object that has nothing above itself in the scene hierarchy is removed because it is no longer part of the scene graph.

This automatic reference counting is usually all you ever need to use. However, the routines **csObject::Ref()**, **csObject::Unref()**, and **csObject::GetRefCount()** allow you to increment, decrement, and retrieve the reference count of a **csObject** should you wish to do so.

Runtime Typing

Each **csObject** knows what type it is. Applications can find out the class object of an instance by querying the object with **getClassType()**, as in the following example:

```
// csContainer *ctr

    if(ctr->getType() == csMaterial::getClassType())
        printf("It's a csMaterial!\n");
    else
        printf("It's not a csMaterial!\n");
```

You need to know the runtime type of an object so you can invoke the right code to manipulate an object.

For checking the derivation of a type, use **csObject::isOfType()**.

The csContainer Class

csContainer objects contain data associated with scene graphs. The data in **csContainer** objects is grouped into fields (**csField**). Fields are not accessible directly to applications. Instead, **set()** and **get()** methods are provided to set and return field values.

Each field contains either a single value of a simple data type, such as a float, or a group of values, all of simple data types.

As an abstract, base class, **csContainer** provides functionality common to all objects containing fields, such as generic access to the fields, creating and deleting field connections, and managing reference counts when objects are added and removed as fields.

The csField Class

Fields contain the data of **csContainer** objects; data generally associated with scene graphs. All publicly-accessible fields in classes derived from **csContainer** should be derived from **csField**.

Fields differ from standard C++ data members. Fields are not evaluated until they are queried. Consequently, none of the meta information (for example, the field's name) exists unless you ask for it.

Field Access

Fields are compact but they still allow applications complete access in two ways:

- Indirect access. Methods for field access and modification are part of each class. Most of the time, applications access fields using these **get*()** and **set*()** functions.
- Generic access. Applications can query any container object abstractly using **getFieldInfo()** on any object that inherits from **csContainer.** This is useful for getting information about unknown objects and makes it possible, for example, to create a GUI for an application.

Single-Item and Multi-Item Fields

Each field contains either a single value of a simple data type, such as a float, or a group of values, all of simple data types.

- Single Item Fields—Single-valued field types, including **SFDouble**, **SFEnum**, **SFRef**, **SFString**, **SFInt**, **SFFloat**, **SFVec2f**, **SFVec3f**, **SFVec4f**, **SFBitMask**, **SFName**, **SFMatrix4f**, **SFRotation**.
- Multiple Item Fields—Multi-valued field types, including **MFRef**, **MFString**, **MFInt**, **MFFloatMFMatrix4f**, **MFVec2f**, **MFVec3f**, **MFVec4f**, **MFRotation**.

For more information about single- and multi-item fields, see "Setting the Values in Scene Graph Nodes" on page 61.

The csNode Class

All Cosmo 3D scene graph components, except leaf objects, such as **csGeometry**, are derived from **csNode**. **csNode,** a subclass of **csContainer**, maintains a bounding sphere for the geometry and the descendant geometry associated with a csNode-type object, such as **csGroup**.

csNode is the fundamental object to which **csAction**s are applied.

For more information about leaf objects, see "Leaf Nodes" on page 57.

For more information about bounding spheres, see "Bounding Volumes" on page 187.

Scene Graph Construction Classes

This section discusses several essential elements of a scene graph.These elements are part of most scene graphs and make it possible for the geometry elements of a model to be drawn and to relate to one another.

Figure 1-1 shows a basic scene graph similar to the example program discussed in Appendix B, "Cosmo 3D Sample Application." Black lines indicate parent-child relationships; gray lines indicate class-field links.

The figure shows the following elements, which are discussed in this section:

- "The csGroup Class"—allows you to group **csNode**s.
- "The csTransform Class"—applies a transformation, such as rotation, scaling, to all its children.
- "The csShape Class"—encapsulates a geometric shape; providing appearance and geometry fields and a **draw()** method.
- "The csAppearance Class"—contains fields to specify the material properties of a surface, including transparency, color, and texture.
- "The csGeometry Class"—encapsulates the geometric data to which a **csAppearance** can be applied.

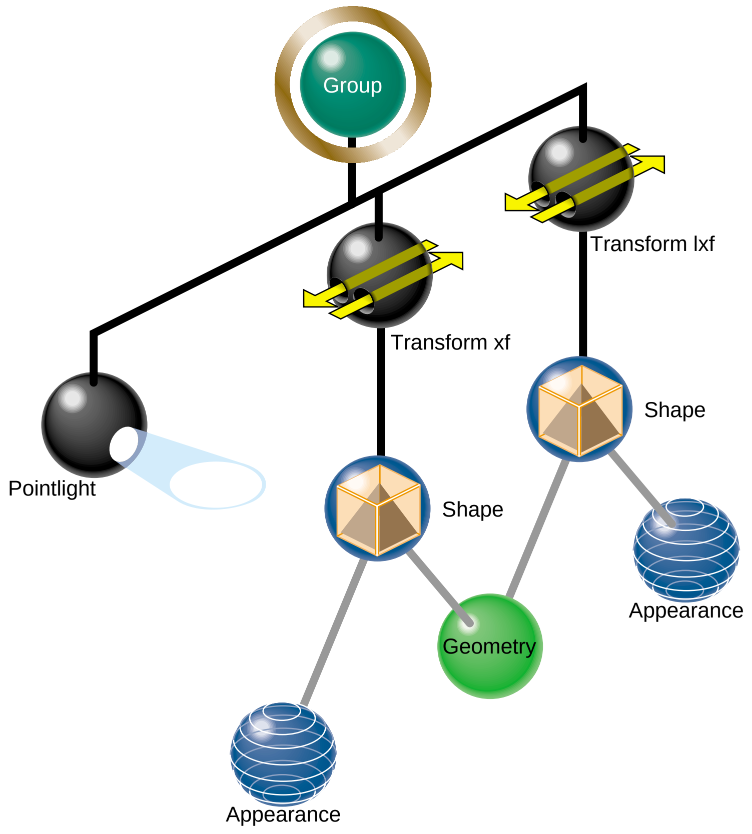

Figure 1-1 Cube Scene Graph

The csGroup Class

The **csGroup** class allows applications to group a list of **csNode**s. When the application then applies actions to the **csGroup**, the actions traverse the scene graph starting at the group-type node. The group-type node passes the action to some or all of its children. The bounding sphere of a **csGroup** is the bounding sphere containing all the bounding spheres of its children.

In Figure 1-1, a group node is the top of the scene graph, joining two **csTransform** nodes and a light.

The csTransform Class

A **csTransform** is a **csGroup** that allows applications to apply a transformation to all of its children. A **csTransformAction** pushes down an action's matrix stack, applies the transform to the top of the stack, visits the children, and then pop the action's matrix stack.

See "Transforming Shapes to New Locations, Sizes, and Orientations" on page 79 for more information.

Once you define the orientation of a shape, you use **csTransform** nodes to place and orient the shape in a different coordinate system. *World space* is the coordinate system of the root node. If all the shapes in a scene graph are transformed into world space, a **csCamera** object attached to the root node can view all the shapes in the scene graph together in one coordinate system.

World space is rendered when a draw action is applied to the root node of the scene graph; local space is rendered when a draw action is applied to a subsection of the scene graph. The same object rendered in these two spaces may appear different, for example, a shape in world space may appear smaller than in local space because it is farther from the viewer; it might also be rotated and positioned differently.

There are usually many transformation nodes in a scene graph and a shape is often transformed more than once. Figure 1-2 illustrates how a leaf node is first transformed twice, then placed in world space.

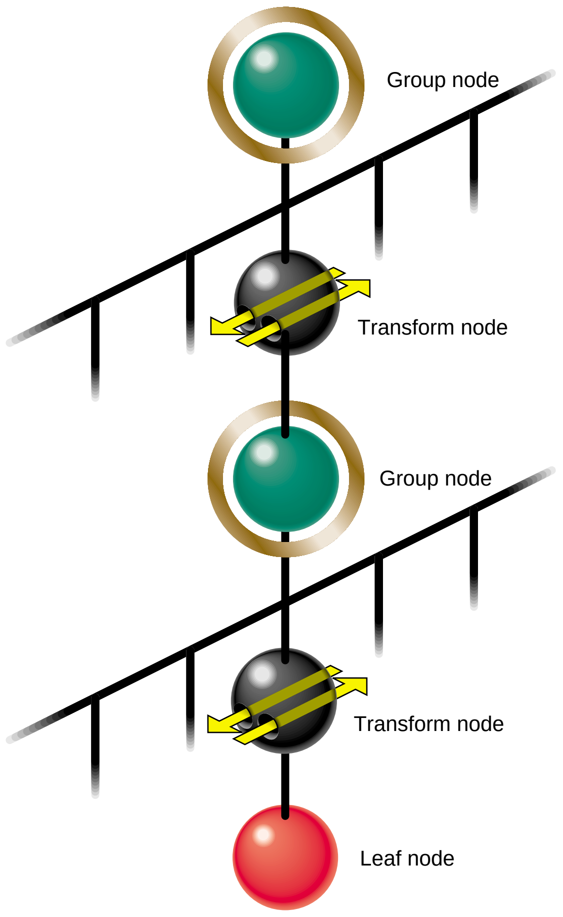

Figure 1-2 Two Transformations into World Space

The csShape Class

csShape nodes, derived from **csNode**, define a textured geometry by associating a **csAppearance**, which describes the look of a shape (such as its color), with a **csGeometry**, which defines the dimensions of the geometry (such as whether the geometry is a cube or sphere).

The csAppearance Class

A **csAppearance** contains fields to specify the material properties of a surface, including transparency, color, and texture. **csAppearance** also provides some facilities borrowed from OpenGL, like specifying whether the surface is drawn filled or in wireframe mode, and the alpha and depth functions to use. **csAppearance** is associated with a **csGeometry** container by a **csShape**, which contains fields for one appearance and a list of geometry.

The csGeometry Class

csGeometry encapsulates the geometric data to which a **csAppearance** can be applied. For example, a **csGeometry** can define a sphere onto which the texture of an orange can be applied to create a realistic image of an orange. Together, **csGeometry** and **csAppearance** combine to form a textured shape. **csShape** associates the two classes.

Classes That Determine How Things Are Drawn

The set of nodes discussed in this section determines how things are drawn.

- "csContext"—Maintains the OpenGL state, for example,
- "The csEnvironment Classes"**—**Determines how lights and fog are applied to its children.

csContext

csContext defines the default, global graphics state of shapes in the scene graph. Shapes inherit some or all of the **csContext** values according to the values set in a mask. **csAppearance** values set on shapes override the default **csContext** values. Similarly, geometry values, defined in **csContext**, can be overridden by individual shapes.

csContext is multi-threaded. A thread can associate a **csContext** and a **csWindow** to facilitate multi-threaded processing. For more information about multi-threading, see Chapter 13, "Multiprocessing."

For more general information about **csContext**, see "csContext Overview" on page 35.

The csEnvironment Classes

The **csEnvironment** class determines how lights and fog are applied to the scene graph.

The lights that affect a **csShape** during a **csDrawAction** are the lights attached to all ancestor **csEnvironments** of the shape, plus all lights applied before invoking the traversal. For more information on lights, see Chapter 8, "Lighting and Fog."

Classes defining Geometric Objects

The actual geometric objects in a Cosmo 3D scene graph are derived as follows:

- As a direct subclass of **csGeometry**. These subclasses include **csCone**, **csSphere**, and so on.
- A **csGeoSet** is a collection of primitives, such as points, lines, triangles, and triangle strips, that, when arranged, create a geometry.

For more information, see Chapter 2, "Creating Geometries."

Steps for Creating and Displaying a Simple Scene Graph

The following procedure summarizes the steps you take to create and render the simple scene graph shown in Figure 1-1. This scene graph is created by the example program discussed in Appendix B, "Cosmo 3D Sample Application."

1. Create **csAppearance** and **csGeometry** containers to define the appearance and the geometry of a shape.

   For more information on setting **csAppearance** values, see Chapter 3, "Specifying the Appearance of Geometries." For more information on setting **csGeometry** values, see Chapter 2, "Creating Geometries."

2. Create **csShape** and **csTransform** nodes.

   For more information on setting **csShape** values, see Chapter 2, "Creating Geometries."

3. Associate the **csAppearance** and **csGeometry** containers using the **csShape** node.

4. Add the **csShape** node as a child of a **csTransform** node.

   The **csTransform** node orients and positions the geometry encapsulated in the **csShape** node. For more information on setting **csTransform** values, see Chapter 6, "Placing Shapes in a Scene."

   Note: A **csShape** node by itself can be a complete scene graph. Typically, however, scene graphs have many **csShape** nodes, most of which are connected to other parts of the scene graph with **csTransform** nodes.

5. Add the **csTransform** node as a child of a **csGroup**-type node.

   For more information about adding nodes to scene graphs, see Chapter 6, “Placing Shapes in a Scene.”

6. Create a window, **csWindow**, in which to view the application and interact with it.
7. Set the current graphical context, **csContext**.
8. Draw all of the shapes in world space by applying a **csDrawAction** to the root of the scene graph.

   The root node is the **csGroup**-type node at the “top” of the scene graph. For more information about draw actions, see Chapter 7, “Traversing the Scene Graph.”

Chapter 2

Creating Geometries

csGeometry is an abstract class. All derivations of the class represent one or more geometric objects, either concrete (such as a sphere or cube) or abstract (such as **geoSet**). The appearance of a shape—whether a sphere is dotted or striped— is characterized by a **csAppearance** object, **csContext** object, or both. Combining a geometry with an appearance completely describes the graphic content of a rendered object.

A **csGeoSet** is a collection of primitives, such as points, lines, triangles, and triangle strips, that, when arranged, create a geometry. For example, a collection of points can represent a star field and a collection of triangles can be arranged to form a sphere or a landscape.

After a brief terminology overview, the first part of this chapter discusses the ready-made geometries available in Cosmo 3D, such as **csSphere** and **csCube**. The remainder of the chapter discusses how to create your own **csGeoSet**-derived classes and how to use the **csGeoSet**-derived classes provided by Cosmo 3D.

These are the sections in this chapter:

Geometry Terminology

Table 2-1 briefly summarizes the geometry terminology used in this manual. Understanding the key terms will help you understand the discussions of the different elements.

Table 2-1 Geometry Terminology

Term	Description	Encapsulated in
Geometry	An object of any form; the surface of which is uniform and non-descript.	**csGeometry** objects or objects derived from this class.
Appearance	Contains all the parameters that specify the look of a geometry.	**csAppearance** object.
*S*hape	Combination of a geometry and an appearance.	**csShape** object.
Context	Maintains and manages the graphics state.	**csContext** object.

Using Large Geometries

Cosmo 3D provides five ready-made geometries:

- csSphere
- csCube
- csBox
- csCone
- csCylinder

Each class has methods that allow you to set and retrieve the values necessary to define the geometry, including (where appropriate)

- coordinates of the center
- length of the radius
- height
- width

The names of the methods that set and retrieve these values are intuitively obvious, for example, to set and retrieve the coordinates of the center of a geometry, you use methods similar to the following:

```
void setCenter(const csVec3f& center);
void getCenter(csVec3f& center);
```

Creating csGeoSet Objects

csGeoSet is a virtual class from which all geometric primitives are derived. Each **csGeoSet**-derived class contains a collection of primitives, such as points, quads, or triangle strips. All of the primitives in a collection are of the same type. You can construct a geometric object by specifying the coordinates of several of these primitives and combine them in a collection. For example, you can arrange triangles to form a sphere or a landscape. The vertices, normals, colors, and texture coordinates of each primitive are captured as attributes of each primitive.

Figure 2-1 illustrates how:

- Each **csGeoSet**-derived object contains an array of primitive shapes.
- Each primitive is made of an array of four attributes.
- Each of the four attributes refers to an array of attribute values, as shown in Figure 2-1.

Note: The order of the attributes can be changed depending on the needs of the application.

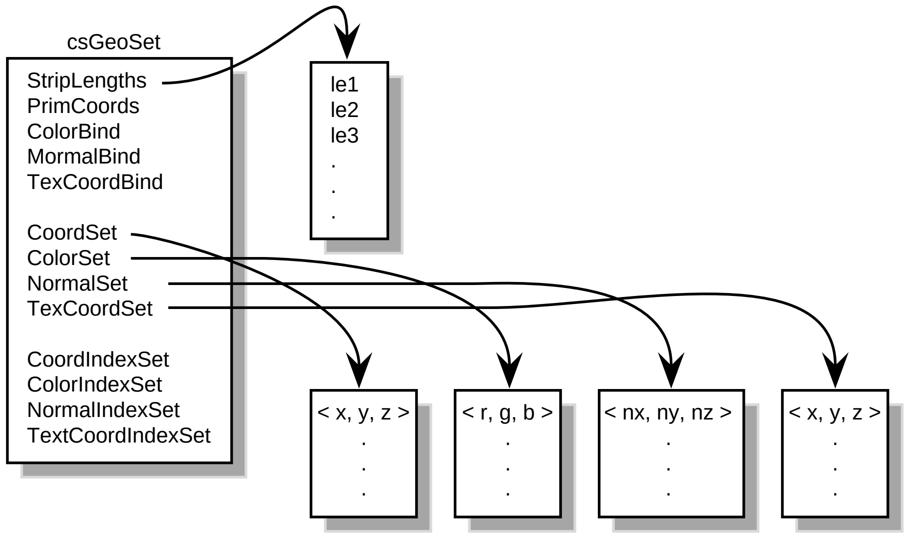

Figure 2-1 Primitives in a csGeoSet

These attributes are captured in **csGeoSet** fields.

csGeoSet Fields

The fields in a **csGeoSet** object can be grouped in the following manner:

Table 2-2 Fields in a csGeoSet

	Field	**Default**
General settings	short cullFace	BACK
	int primCount	0
Attribute specifications	Color colors	NULL
	Normal normals	NULL
	TexCoord texCoords	NULL
	Coord coords	NULL
Attribute index specifications	Index colorIndices	NULL
	Index normalIndices	NULL
	Index texCoordIndices	NULL
	Index coordIndices	NULL
Attribute binding specifications	char colorBind	OFF
	char normalBind	OFF
	char texCoordBind	OFF

The remainder of this section describes **csGeoSet** general settings. The other parts of this chapter describe the attribute fields.

For more information about cull facing, see “Face Culling” on page 152.

Setting the Number of Primitives

The following **csGeoSet** methods affect all of the primitives in a **csGeoSet** object:

```
void setPrimCount(csInt primCount);
csInt getPrimCount();
```

To specify or retrieve the number of primitives in a **csGeoSet**, use the **setPrimCount()** and **getPrimCount()** methods. Appendix B, “Cosmo 3D Sample Application” shows how to retrieve the number of primitives in a **csGeoSet**.

csGeoSet Attributes

csGeoSet is a virtual class from which all geometric primitives are derived. Cosmo 3D-supplied **csGeoSet**-derived classes include, for example:

- **csPointSet**—A collection of equally-sized points.
- **csLineStripSet**—A collection of linestrips, also known as polylines, of equal width.
- **csTriStripSet**—A collection of triangle strips.
- **csPolySet**—A collection of convex, coplanar polygons.

All of the primitives within a given set are equal in size. These primitives are defined by an array of four attributes:

- color—(red, green, blue, alpha)
- normal—(N_x, N_y, N_z)
- texture coordinates—(S, T)
- coordinates—(X, Y, Z)

Each attribute consists of an array of two to four values; a primitive is defined by these twelve values.

Note: Although texture coordinates can be specified using four values (S, T, R, Q), the R value has no current meaning in Cosmo 3D because it does not support textures greater than two dimensions, and the Q value is always one.

Attribute Bindings

Not all attributes can be applied with the same level of specificity. The levels of specificity include

- The entire collection of primitives in a **csGeoSet** object.
- Individual primitives in a **csGeoSet** object.
- Individual vertices of individual primitives in a **csGeoSet** object.

For example, a single color can be specified for the entire collection of primitives, for individual primitives, or per vertex. One set of coordinates, on the other hand, cannot be specified for the entire collection of primitives, cannot be specified for individual primitives, but must be specified per vertex. It does not make sense for all of the primitives in a collection to have the same coordinates, nor does it make sense for all vertices in each primitive to have the same coordinates. Each vertex must have its own coordinates.

Each level of specificity is called a different *binding*, for example, an attribute that is specified for an entire collection of primitives is said to have an OVERALL binding. A binding tells you how many primitives in a **csGeoSet** object an attribute applies to. Table 2-3 shows the different possible bindings.

Table 2-3 Attribute Bindings

	OFF	**OVERALL**	**PER_PRIMITIVE**	**PER_VERTEX**
colors	yes	yes	yes	yes
normals	yes	yes	yes	yes
texture coordinates	yes	no	no	yes
coordinates	no	no	no	yes

All attributes in a **csGeoSet** collection must share the same set of attribute bindings, for example, you cannot specify colors-per-vertex for some primitives and colors-per-primitive for others in the same **csGeoSet** object, the color binding must be the same. You can, however, have, for example, color-per-vertex and overall normal bindings in the same **csGeoSet**.

Setting Attribute Bindings

Three **set...()** methods in **csGeoSet** specify the attribute bindings for a **csGeoSet** object:

```
void setNormalBind(NormalBindEnum normalBind);
void setColorBind(ColorBindEnum colorBind);
void setTexCoordBind(TexCoordBindEnum texCoordBind);
```

There is a corresponding set of **get...()** methods that retrieve the attribute bindings for the normals, colors, and texture coordinates, respectively.

The enumerated binding values that are valid for each of the attributes coincide with the entries in Table 2-3.

```
enum NormalBindEnum
    {
    NO_NORMS,
    OVERALL_NORMS,
    PER_PRIM_NORMS,
    PER_VERTEX_NORMS,
    };
enum ColorBindEnum
    {
    NO_COLORS,
    OVERALL_COLORS,
    PER_PRIM_COLORS,
    PER_VERTEX_COLORS,
    };
enum TexCoordBindEnum
    {
    NO_TEX_COORDS,
    PER_VERTEX_TEX_COORDS
    }
```

To set the color of all the primitives in a **csGeoSet** object to the same value, for example, use the OVERALL_COLORS binding in code similar to the following:

```
csTriangleStripSet* myTriangleStrip = new csTriangleStripSet();
myTriangleStrip->setColorBind(csGeoSet::OVERALL_COLORS);
```

Setting Attributes

Now that you know how to set attribute bindings, you need to know how to set the attributes themselves.

As shown in Figure 2-1, **csGeoSet** objects store their primitives in an array. The array contains:

- Three attribute values in the Normal array.
- Three (or four) attribute values in the Color array.
- Two attribute values in the Texture Coordinate array.
- Three attribute values in the Coordinate array.

This pattern continues, as shown in Figure 2-2.

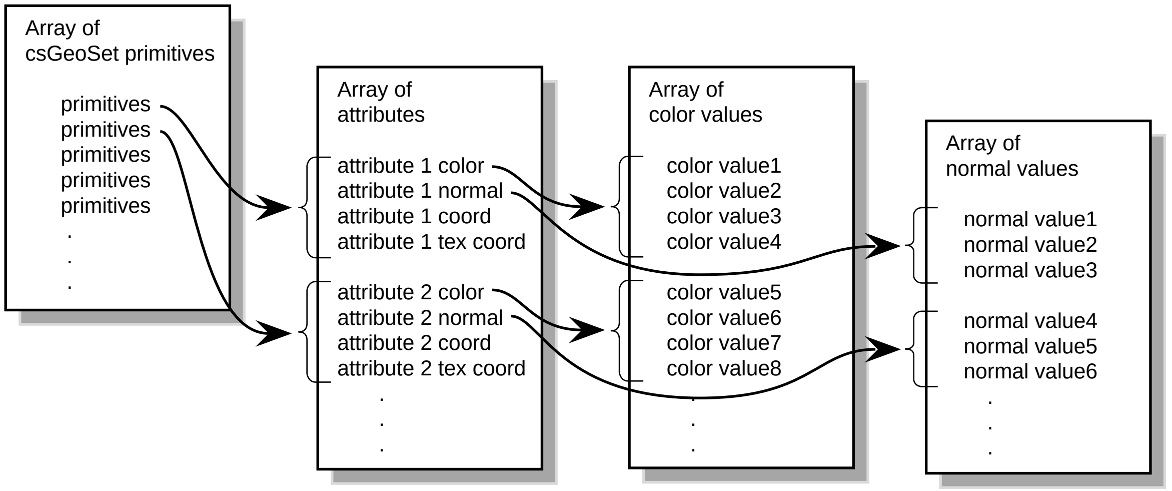

Figure 2-2 Sequential Specification of Attributes Per Primitive

Indexing Attributes

Another option is to index the attribute values so that primitives can access any attribute value and more than one primitive can use the same attribute value, as shown in Figure 2-3.

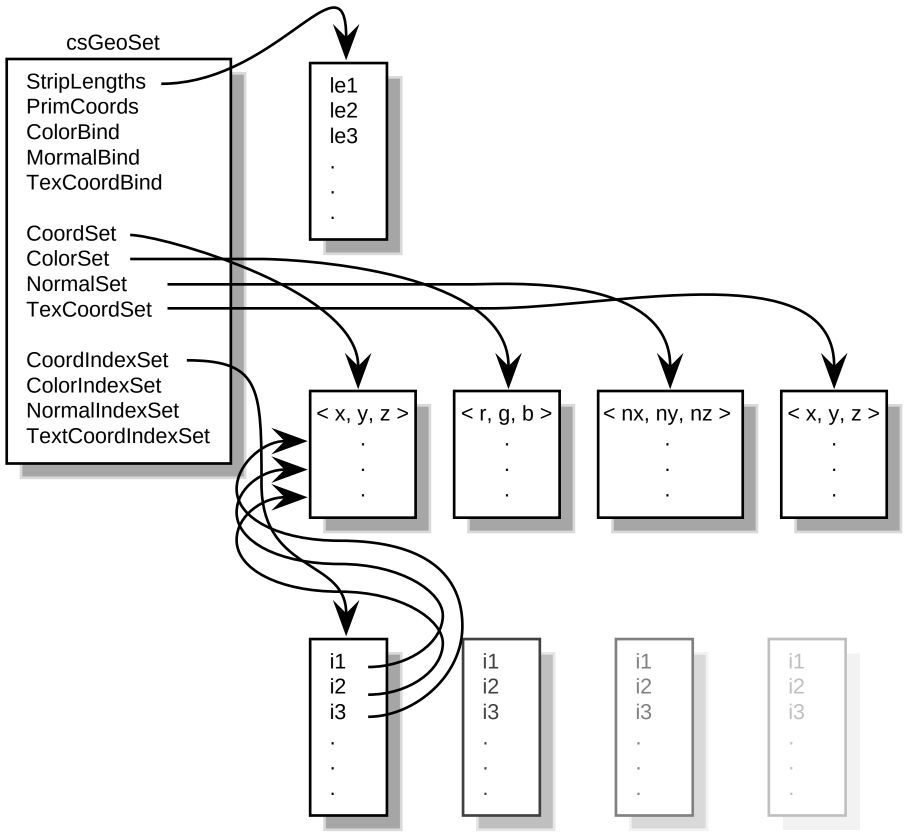

Figure 2-3 Indexed Attributes

When to Index Attributes

For all primitives in a **csGeoSet**, you have to decide whether to use indexed or sequential attributes; that is, all of the primitives within one **csGeoSet** must be referenced either sequentially or by index. You cannot mix the two reference methods.

The governing principle for indexing attributes or not is how many vertices in a geometry are shared. Consider the following two examples in Figure 2-4, where each dot marks a vertex.

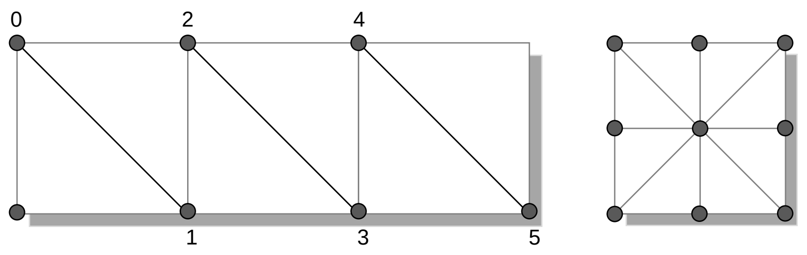

Figure 2-4 Deciding Whether to Index Attributes

In the triangle strip, each vertex is shared by two adjoining triangles. In the square, the same vertex is shared by eight triangles. Consider the task that is required to move these vertices when, for example, morphing the object. If the vertices were not indexed, in the square, the application would have to look up and alter eight triangles to change one vertex.

In the case of the square, it is much more efficient to index the attributes. On the other hand, if the attributes in the triangle strip were indexed, since each vertex is shared by only two triangles, the index look-up time would exceed the time it would take to simply update the vertices sequentially. In the case of the triangle strip, rendering is improved by handling the attributes sequentially.

The deciding factor governing whether or not to index attributes relates to the number of primitives that share the same attribute: if attributes are shared by many primitives, the attributes should be indexed; if attributes are not shared by many primitives, the attributes should be handled sequentially.

"Indexing Attributes" on page 28 describes the methods you use to index attributes.

Specifying Attributes

Whether you index your attributes or not, you must use the following **set...()** methods in **csGeoSet** to specify the attributes in a specific **csGeoSet** object:

```
void setCoordsSet(csCoordSet* coords);
void setNormalsSet(csNormalSet* normals);
void setColorsSet(csColorSet* colors);
void setTexCoordsSet(csTexCoordSet* texCoords);
```

There is a corresponding set of **get...()** methods that retrieve the index settings for the coordinates, normals, colors, and texture coordinates, respectively.

The **set...()** methods have the following arguments:

- *coords* is a three-dimensional array of coordinates representing the coordinates of every vertex in every primitive in a **csGeoSet** object.
- *normals* is a three-dimensional array of normals for potentially every vertex in every primitive in a **csGeoSet** object, depending on the binding.
- *colors* is a four-dimensional array of colors for potentially every vertex in every primitive in a **csGeoSet** object, depending on the binding.
- *texCoords* is a two-dimensional array of coordinates representing the texture coordinates of every vertex in every primitive in a **csGeoSet** object.

Using More Specific Attribute Arrays

Each of the four attributes has its own array. You must use one of the more specifically-defined virtual array classes, as follows:

```
csCoordSet3f();
csNormalSet3f();
csColorSet3f();
csColorSet4f();
csTexCoordSet2f();
```

Each of these null constructors is overridden by a set of two constructors that are similar in form to the following:

```
csCoordSet3f(int n);
csCoordSet3f(csData *array, short offset, short stride);
```

The first constructor allows you to specify the number of array primitives, *n*.

The second constructor allows you to reference an array, **array*, of attribute values, specify the offset, *offset*, if any, and the stride, *stride*.

The stride mechanism that lets an application choose to keep all data staggered in a single array (or use two arrays). For example, you could combine color, vertex, and coordinate data and access each type as needed using the stride number. Stride specifies the byte offset between pointers to consecutive vertexes, in effect, stride is a relative offset, as shown in Figure 2-5.

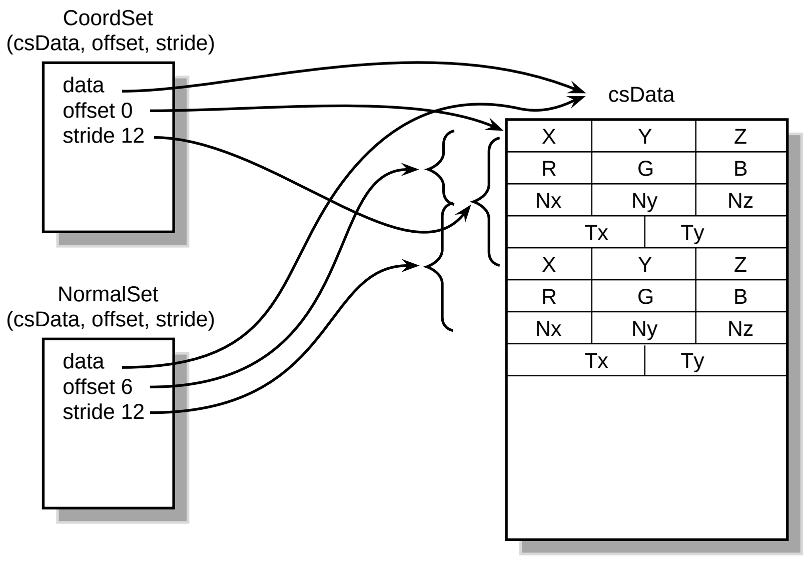

Figure 2-5 Stride and Offset Values

Set and Get Methods

Each of the virtual attribute-array classes, both the general and specific, have set and get methods to set and return the values of the array. All set and get methods use the following form:

```
void setCoordsSet(csCoordSet* coords);
csCoordSet* getCoordsSet();
```

Indexing Attributes

An indexed **csGeoSet** object uses a list of unsigned short integers to index an attribute array. Four **set...()** methods in **csGeoSet** specify these indices:

```
void setCoordIndices(csIndexSet* coordIndices);
void setNormalIndices(csIndexSet* normalIndices);
void setColorIndices(csIndexSet* colorIndices);
void setTexCoordIndices(csIndexSet* texCoordIndices);
```

There is a corresponding set of **get...()** methods that retrieve the index settings for the coordinates, normals, colors, and texture coordinates, respectively.

coordIndices is an array of coordinate indices. Each index points to a member in the coordinate attribute array, as shown in Figure 2-3.

normalIndices is an array of normal indices. *colorIndices* is an array of color indices. *texCoordIndices* is an array of texture coordinate indices.

Setting Attributes Example

Example 2-1 shows how to set attributes and their bindings.

Example 2-1 Setting Attributes

```
// Create a csGeoSet object
csTriStripSet *gset = new csTriStripSet;

// Allocate the attribute arrays
csCoordSet3f        *vset = new csCoordSet3f(NumRings*RingVerts);
csNormalSet3f       *nset = new csNormalSet3f(NumRings*RingVerts);
csIndexSet          *iset = new csIndexSet((NumRings-1) *
                       2 * (RingVerts + 1));
csColorSet4f        *cset = new csColorSet4f(NumRings-1);
csIndexSet          *lengths = new csIndexSet(NumRings-1);

// Set the attributes
gset->setCoords(vset);
gset->setNormals(nset);
gset->setColors(cset);
// Set the attribute indices
gset->setCoordIndices(iset);
gset->setNormalIndices(iset);
gset->setPrimCount(NumRings-1);
```

```
// Set the attribute bindings
gset->setNormalBind(csGeoSet::PER_VERTEX_NORMS);
gset->setColorBind(csGeoSet::PER_PRIM_COLORS);

// Prepare to fill the Attribute and Indices arrays
csVec3f     *coords = vset->coords()->edit();
csVec3f     *norms = nset->normals()->edit();
int         *indices = iset->indices()->edit();
```

Editing Attribute Arrays

Cosmo 3D allows you to modify the values in arrays using the **csNormalSet3f::edit()** and **csNormalSet3f::editDone(**) methods. Although you can modify the values, you cannot change the number of values in the array.

edit() returns a pointer to the attribute array. **editDone()** notifies any engines or sensors connected to this field that the array has changed.

It is illegal to call any other editing methods between **edit()** and **editDone()**.

Example 2-2 shows an example of editing attribute arrays.

Example 2-2 Editing Attribute Arrays

```
// cube normals
    csNormalSet3f *nset = new csNormalSet3f(numCubeNorms);
    nset->vector()->edit();
#if 0
    for (i=0; i<numCubeNorms; i++)
        nset->vector()->set(i,
        csVec3f(cubeNorms[i][0], cubeNorms[i][1], cubeNorms[i][2]));
#else
    nset->vector()->setRange(0, numCubeNorms, (csVec3f *)cubeNorms);
#endif
    nset->vector()->editDone();

    gset->setNormalSet(nset);
```

Cosmo 3D-Derived csGeoSet Objects

Cosmo 3D provides the following **csGeoSet** collections. Each is a derivative of **csGeoSet**.

- **csPointSet**—A collection of equally-sized points.
- **csLineSet**—A collection of lines of equal length.
- **csIndexedLineSet—**A set of indexed line strips.
- **csLineStripSet—**A collection of linestrips, also known as polylines.
- **csTriSet—**A collection of triangles.
- **csTriFanStrip—**A collection of triangles that share a common vertex.
- **csTriStripSet—**A collection of triangle strips.
- **csPolySet**—A collection of convex, coplanar polygons.
- **csQuadSet—**A collection of quadrilaterals.
- **csIndexedFaceSet—**A polygon with faces that are indexed.

The following sections describe each of these primitive collections.

All of the classes contain virtual **draw()** and **calcBound()** methods. The **draw()** method specifies how a **csGeoSet** object is drawn. The **calcBound()** method specifies how the bounding box is computed. Other fields are specific to their geometries.

Using csPointSet

A **csPointSet** object contains a collection of equally-sized points. Point size is the diameter of each point in pixels.

csPointSet contains the following fields:

```
void    setSize(csFloat size);
csFloat getSize();
```

The **setSize()** and **getSize()** methods allow you to specify and find out, respectively, the diameter, in pixels, of all the points in a **csPointSet** object.

Using csLineSet

A **csLineSet** object contains a collection of lines of equal length. The fields allow you to set and return the width of the lines used for drawing.

```
void        setWidth(csFloat width);
csFloat     getWidth();
```

Using csIndexedLineSet

A **csIndexedLineSet** object contains an indexed collection of lines of equal length. The fields allow you to set and return the colors of the lines in the collection.

```
csMFInt*    coordIndex() const;
csMFInt*    colorIndex() const;
void        setColorPerVertex(csBool colorPerVertex);
csBool      getColorPerVertex();
```

Using csLineStripSet

A **csLineStripSet** object contains a collection of linestrips, otherwise known as polylines, of equal width. Line width is specified in pixels.

csLineStripSet contains the following fields:

```
csMFInt* stripLength () const

void        setWidth(csFloat width);
csFloat     getWidth();
```

The**StripLength()** method allows you to specify and find out, respectively, how many line segments are in a **csLineStripSet** object.

The **setWidth()** and **getWidth()** methods allow you to specify and find out, respectively, the width, in pixels, of each linestrip in a **csLineStripSet** object.

Using csTriSet

A **csTriSet** object contains a collection of triangles. This class serves as a class from which **csTriFanSet** and **csTriStripSet** are derived.

Using csTriFanSet

A **csTriFanSet** is a set of triangles all of which share one common vertex, as shown in Figure 2-6.

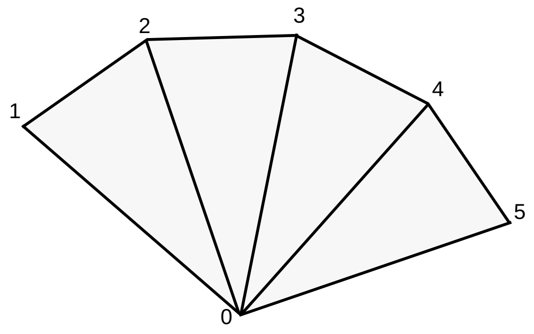

Figure 2-6 TriFanSet

You use the following method to retrieve or set the number of triangles in the **csTriFanSet**.

```
csMFInt*    fanLength() const;
```

Using csTriStripSet

A **csTriStripSet** object contains a collection of triangle strips. A triangle strip is a series of adjacent triangles that form a strip, as shown in Figure 2-7.

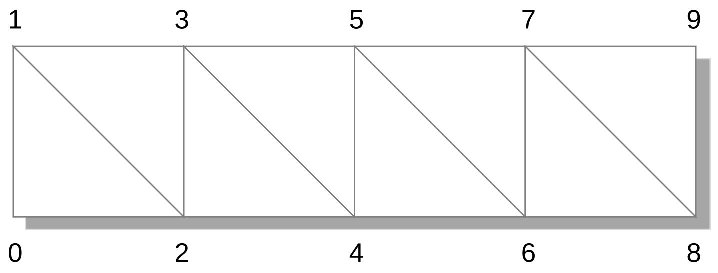

Figure 2-7 Triangle Strip

csTriStripSet contains the following field:

```
csMFInt* stripLength() const;
```

This field allows you to specify and find out how long each triangle strip is in a **csTriStripSet** object. The length is expressed in the number of vertices per strip, for example, three tristrips with individual lengths of 4, 6, and 8, would be represented by an array of three integers:

```
csMFInt* length = [4, 6, 8];
```

Using csPolySet

A **csPolySet** object contains a collection of polygons. Polygons may have different numbers of sides but must be convex and coplanar.

csPolySet contains the following methods:

```
csMFInt* polyLength () const;
```

This field allows you to specify and find out how many sides there are per polygon in a **csPolySet** object.

Using csQuadSet

A **csQuadSet** object contains a collection of quadrilaterals.

Using csIndexedFaceSet

A **csIndexedFaceSet** object contains a collection of polyhedrons of equal size. The member functions allow you to set and return the size of the polyhedrons in the collection.

```
csMFInt*    coordIndex() const;
csMFInt*    colorIndex() const;
csMFInt*    normalIndex() const;
csMFInt*    texCoordIndex() const;

void        setCCW(csBool ccw);
void        setSolid(csBool solid);
void        setConvex(csBool convex);
void        setCreaseAngle(csFloat creaseAngle);
void        setColorPerVertex(csBool colorPerVertex);
void        setNormalPerVertex(csBool normalPerVertex);
```

There is a corresponding **get...()** method for every **set...()** statement.

The first four fields contain arrays for storing the color.

setCCW() is true if the vertices of these faces wind counter-clockwise when viewed from the front.

setSolid() is true if this set of faces forms a closed volume (“solid”); in that case, faces on the side of the solid facing away from the viewpoint don’t need to be drawn.

setConvex() is true if the faces in this set are convex. (Currently ignored.)

setCreaseAngle() sets the crease angle. If the angle between two faces is more than the crease angle, the faces are assumed to be part of a single surface and are smooth shaded. (Currently ignored.)

coordIndex() sets a VRML 2.0-style vertex coordinate index set.

colorIndex() sets a VRML 2.0-style color index set.

texCoordIndex() sets a VRML 2.0-style texture coordinate index set.

normalIndex() sets a VRML 2.0-style normal index set.

setColorPerVertex() is true if colors are assigned per vertex, otherwise per face.

setNormalPerVertex() is true if normals are assigned per vertex, otherwise per face.

Chapter 3

Specifying the Appearance of Geometries

The geometry and appearance of a shape are independent of one another. The appearance of a shape is the two-dimensional texture, such as the rind of an orange, that is mapped onto a geometry.

This chapter describes how to specify the appearance of a geometry in the following sections::

csContext Overview

A **csContext** object maintains the OpenGL graphics state for a scene graph and therefore contains all the default appearance values necessary to render a shape.

Appearance values, such as material and texture, can be specified per shape using **csAppearance**. If **csAppearance** fields are not set, the shape inherits the default appearance values set in **csContext**. For optimal performance, set as few **csAppearance** object fields as possible by setting the global defaults in **csContext** to values that satisfy the majority of geometries in a scene graph. This practice minimizes state changes while rendering.

There is an inheritance mask in each **csAppearance** that specifies which appearance values are inherited by **csAppearance** from **csContext**. **csAppearance** values automatically override **csContext** default values on a per-shape basis, regardless of the bit values in the inheritance mask.

State Machine

csContext maintains and manages OpenGL graphics state for the purpose of efficient graphics pipeline state control. OpenGL is a state machine: you put it into various states (or modes) that then remain in effect until you change them. For example, the current color is a state variable. The Cosmo 3D context maintains two notions of state:

- Default state—is the global graphics state defined on a per-context basis and maintained separately from the current state.
- Current state—represents the accumulation of the default state and the state set when **csAppearance** nodes are encountered during a traversal.

State that is not explicitly set in a **csAppearance** via the appropriate **csAppearance set()** methods is inherited from the default state. An inheritance mask, however, specifies which **csContext** fields a shape inherits by default.

The effect of a **set()** method is immediate, as if you made the OpenGL calls directly. The **set()** methods affect the current state but have no effect on the default state.

Inheritance Mask

Inheritance masks specify which csContext fields are inherited by csAppearance. Each bit in the bitmask corresponds to a specific csAppearance field. All fields are inherited by default.

Figure 3-1 demonstrates how the inheritance mask works as a result of the code in Example 3-1.

Example 3-1 Inheritance Mask

```
csContext *ctx = new cscontext;

ctx->setCullFace(BACK);
ctx->setLightEnable(TRUE);
.
```

```
.
.
csAppearance *app = new csAppearance;
app->setMaterial(mtl);
shape->setAppearance(app);
```

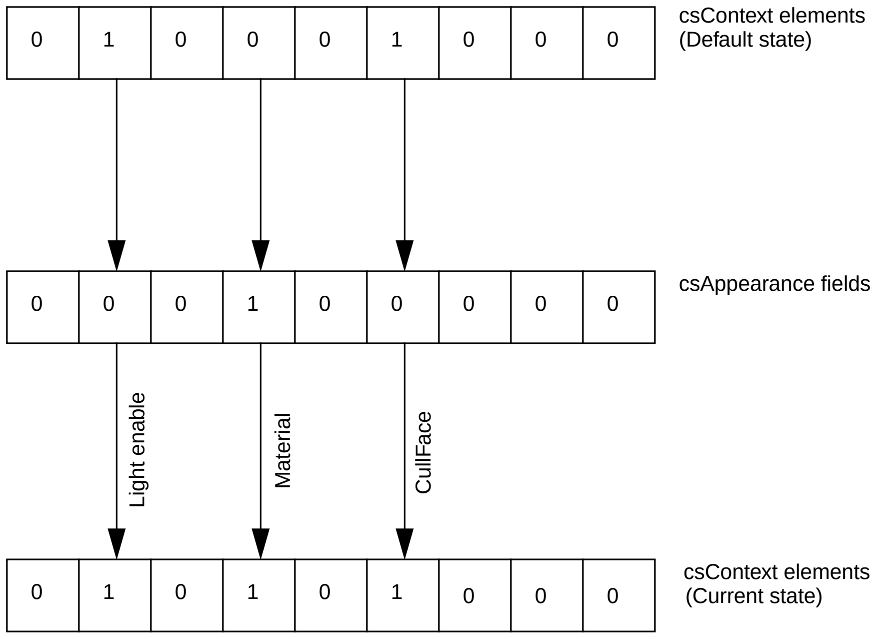

Figure 3-1 Inheritance Mask

All 0 bits indicate that the default value is used.

When you set a **csAppearance** value, its corresponding bit in the inheritance mask is set.

Accessing States

Applications can access the current state using the various **get()** methods. The primary use of the **get()** method is to ask the system about its current state. Cosmo 3D makes this state available so that a user callback can know the current state of the system and make OpenGL calls appropriately.

Warning: It is critical that such a callback not alter the OpenGL state.

You can avoid altering the OpenGL state either by using **csContext**::**set()** calls or by saving and restoring OpenGL state explicitly upon entry and exit of the callback.

What Modifies the Graphics State?

As a **csDrawAction** traverses the scene graph, the current state is modified when:

- Appearances a their draw methods are invoked.
- Calls users make to various **set()** methods in **csContext** invoke pre- and post-node callbacks.

Traversal Order

The actual appearance of a **csGeometry** being drawn is independent of traversal order. Only that **csShape**'s appearance and the default state affect the actual appearance. At no time does Cosmo 3D depend on the traversal order of the draw action. No guarantees are made about traversal order because the traversal order of the draw is subject to change. Because applications cannot depend on draw traversal order to imply the state of the context, methods to query the context for its current state are available.

csContext in Multi-threaded Programs

csContext can be used in a multi-threaded program. When **makeCurrent()** is called within a given thread, that context is attached to the thread. Binding a thread to a context allows multi-pipe or multi-window rendering in parallel using a single, shared copy of the scene graph.

For more information about multi-threaded implementation, see Chapter 13, "Multiprocessing."

Overriding Appearances and Geometry Properties with csContext

In general, **csAppearance** settings override **csContext** default values. You can, however, override **csAppearance** settings using **csContext::pushOverrideAppearance()**. Only one override appearance per context can be in place at a time.

You can override some properties that are not in **csAppearance** and are geometry-specific through the use of **pushOverrideGeoProp(**). Currently, **csCullFace**, **csLineWidth**, and **csPointSize** are the only geometry-specific properties that can be overridden.

Making the Screen One Color

To change the screen to a specified color, use one of the following **csContext** methods:

```
static void clear(int which);
static void clear(int which, float r, float g, float b, float a);
```

where *which* is a bitmask specifying whether to clear the color planes, depth planes, or both.

The first **clear()** method clears the screen to black. The second version allows you to set a uniform color and transparency.

Changing the Context

You can create multiple **csContext** objects but only one can be active at a time. In this way, in addition to changing field values in a context object, you can change the entire context all at once using **makeCurrent()**. This method replaces the current context with the **csContext** object specified in the argument, for example:

```
csContext* context1 = new csContext;
csContext* context2 = new csContext;

context1->makeCurrent(display, window);
context2->makeCurrent(display, window);
```

In this example, the second context replaces the first.

- *display* is a pointer to the X window display.
- *window* is the GLXDrawable in which the scene is displayed.

getCurrent() returns the context object on top of the context stack.

csWindow has a context and calls **makeCurrent()** automatically.

Using csAppearance

csAppearance fields define the appearance of a **csGeometry** object, for example, its texture, material, or color. All of the fields in **csAppearance** are replicated in **csContext**.

Inheriting Appearance Values

To specify the appearance of a **csGeometry**, you can either

- Set all of the appearance fields in a **csAppearance** object.
- Use the inherited, global, default values from the current context, **csContext**.
- Use a combination of the first two options.

If you set all of the fields of an appearance object, the appearance object becomes the full graphic context of the **csShape**. The more appearance fields you set, however, the slower the application's performance because you are triggering lots of state changes.

For maximum performance, set the appearance values in **csContext** to satisfy the maximum number of shapes so that the fewest number of **csAppearance** fields are set on a per-shape basis.

Setting Appearance Fields Locally

The only fields that you should set locally are those that change often, such as the field values for material and texture. Changing a field value locally overrides any value inherited from **csContext**.

The **csAppearance** class includes a series of **set...()** methods to define the appearance characteristics of a geometry. A series of corresponding **get...()** methods provide access to those values. The following **set...()** methods are described in greater detail in the rest of this chapter:

```
void setTexture(csTexture* texture);
void setTexEnable(csBool texEnable);
void setTexMode(csContext::TexModeEnum texMode);
void setTexBlendColor(const csVec4f& texBlendColor);
void setTexBlendColor(csFloat v0, csFloat v1, csFloat v2, csFloat v3);
void setTexEnv(csContext::TexEnvEnum texEnv);
void setTexGen(csTexGen* texGen);
void setTexGenEnable(csBool texGenEnable);

void setMaterial(csMaterial* material);
void setLightEnable(csBool lightEnable);
void setShadeModel(csContext::ShadeModelEnum shadeModel);
void setTranspEnable(csBool transpEnable);
void setTranspMode(csContext::TranspModeEnum transpMode);
void setAlphaFunc(csContext::AlphaFuncEnum alphaFunc);
void setAlphaRef(csFloat alphaRef);
void setBlendColor(const csVec4f& blendColor);
void setBlendColor(csFloat v0, csFloat v1, csFloat v2, csFloat v3);
void setSrcBlendFunc(csContext::SrcBlendFuncEnum srcBlendFunc);
void setDstBlendFunc(csContext::DstBlendFuncEnum dstBlendFunc);
void setColorMask(const csVec4ub &colorMask);
void setColorMask(csUByte v0, csUByte v1, csUByte v2, csUByte v3);
void setDepthFunc(csContext::DepthFuncEnum depthFunc);
void setDepthMask(csUInt depthMask);
void setFogEnable(csBool fogEnable);
void setPolyMode(csContext::PolyModeEnum polyMode);
```

These method are separated into two groups:

- Methods containing the string “tex” modify textures.
- The remaining methods modify the appearance of geometries.

Lazy Updating of Appearance Values

csAppearance values are updated in a lazy way: a value is changed only when it is used. For example, if a ball is currently displayed and you change its color using **setColor()**, the ball would change color immediately on the screen. If, however, the ball is out of view of the camera, the color of the ball would not be updated until it is seen by the camera.

Applying Textures to Geometries

One way to affect the appearance of a geometry is to apply a texture to it. A texture is a rectangular 2D image, for example, a 2D map of the world. This rectangular texture is scaled or repeated to fit on the surface of a 3D object, such as a sphere. The clamping and repetition of a texture over a surface is programmatically controllable.

To create the image of an orange, for example, you first create the orange, pitted texture of orange rind and then apply it to a sphere. The difference between using and not using a texture, in this example, is the difference between rendering a generic sphere and a realistic-looking orange.

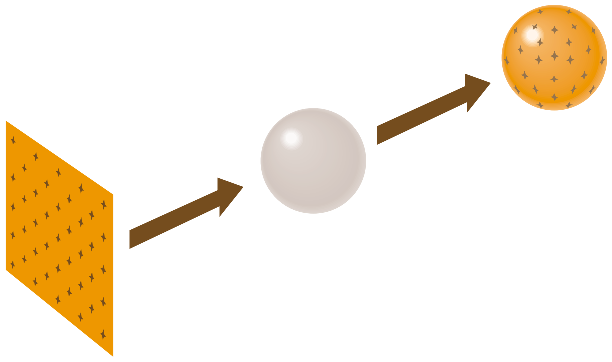

Figure 3-2 Applying a Texture to a Geometry

Texture Map Coordinates

A texture map is always defined by the coordinates s, for the horizontal component, and t, for the vertical component, each of which range in values from 0.0 to 1.0, as shown in Figure 3-3.

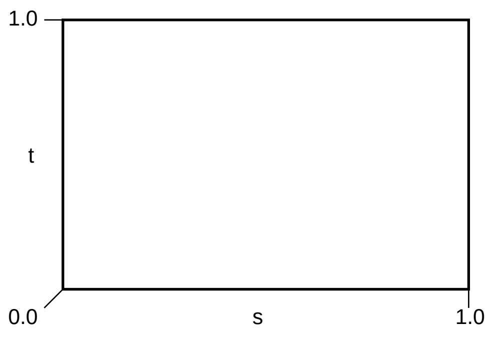

Figure 3-3 Texture Coordinates

Texture coordinates are assigned to each vertex of a geometry either by you or by Cosmo 3D.

Applying a Texture

To apply a texture to a geometry, set the argument of the **csAppearance::setTexEnable()** method to ON. If you do not want to apply a texture to a geometry, set the argument of **setTexEnable()** to OFF.

Texture rendering uses the texture values specified in **csContext** by default. To set the texture values locally, however, use the following methods in **csAppearance**:

setTexture()	to specify the image used as the texture.
setTexMode()	to specify the speed and quality of the rendered texture.
setTexBlendColor()	to specify the color to use in “blend” mode.
setTexEnv()	to specify how texture colors are blended with the colors of a geometry.
setTexGen(), **setTexGenEnable()**	to generate, if enabled, texture coordinates automatically instead of using the csGeoSet’s TexCoordSet.

The following sections describe these methods.

Specifying a Texture Image

To apply a texture to a geometry, supply a **csTexture** object in the argument of **setTexture()**. **csTexture** is a class consisting of the following fields and default values:

```
csSFString filename “noName”
csMFRef  imageLevel[]
csSFEnum format 0
csSFEnum repeat_S REPEAT
csSFEnum repeat_T REPEAT
csSFEnum minFilter FAST
csSFEnum magFilter FAST
csSFEnum source 0
```

imageLevel is an array of MIPmap levels for this texture of type csImages. If this field is not set or has all NULL values, the texture is loaded from csSFString *fileName* instead.

format is the pixel format of the image. For more information about pixel format, see “Color Components” on page 47.

repeat_S and **repeat_T** specify whether or not the texture is repeated in the s and t directions on the geometry, respectively. If the texture is not repeated, it is clamped.

minFilter specifies what to do with texels that project smaller than a screen pixel. Possible values include NEAREST_MIN, LINEAR_MIN, or MIPMAP.

magFilter specifies what to do with texels that project larger than a screen pixel. Possible values include NEAREST_MAX or LINEAR_MAX.

Texture Mode Settings

The texture mode method allows you to specify texture rendering speed, quality, and perspective where speed and quality, and speed and perspective are trade-offs.

You specify the mode of the texture rendering using **setTexMode()** with one of the following arguments from **csContext::TexModeEnum**:

FAST_TEX	for a low quality, more quickly-rendered texture.
NICE_TEX	for a high quality, more slowly-rendered texture.
NON_PERSP_TEX	for a non-perspectively-correct, more quickly-rendered texture.
PERSP_TEX	for a more slowly-rendered texture in perspective.

When you choose the NON_PERSP_TEX mode, Cosmo 3D applies the texture to a geometry without proper perspective. For example, if you apply a texture to a plane extending into the Z dimension, the pattern should not distort but just appear to recede into the distance. In NON_PERSP_TEX mode, however, the pattern is distorted, as shown in Figure 3-4.

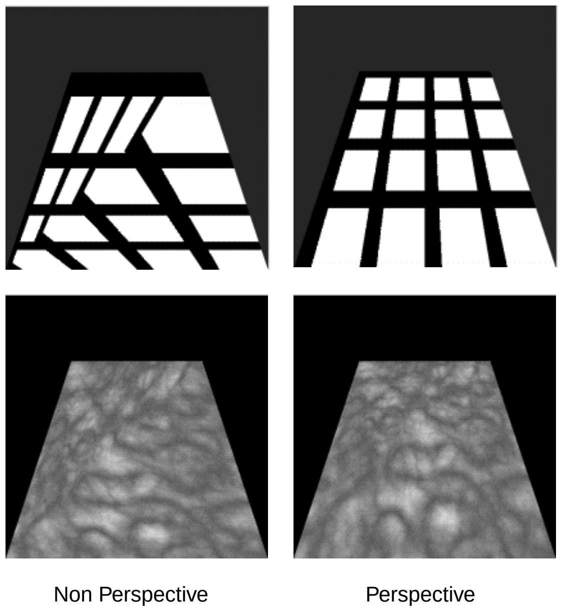

Figure 3-4 Non-Perspective and Perspective Modes

If you enable texture rendering but do not set the texture mode in a **csContext** or **csAppearance** object, the texture rendering mode is defined by the **csTexture** object in the argument of **csAppearance**::**setTexture()** or **csContext**::**setTexture()**. A **csTexture** object can specify one of the values in TexModeEnum.

Texture Environment Settings

Texture environment variables specify how texture colors are blended with the colors of a geometry; the texture color can replace, blend with, or subtract from the colors already on the geometry.

To specify how texture colors are blended with the colors of a geometry, use the **setTexEnv()** method with one of the following **csContext::TexEnvEnum** values as an argument:

MODULATE_TENV	multiplies the shaded color of the geometry by the texture color. If the texture has an alpha component, the alpha value modulates the geometry's transparency, for example, if a black and white texture, such as text, is applied to a green polygon, the polygon remains green and the writing appears as dark green lettering. MODULATE is the default value.
BLEND_TENV	uses the texture color to blend together the blend color and the underlying geometry's color. In the above example, the lettering would be a mixture of green, white, and black.
REPLACE_TENV	replaces the underlying geometry's color with the texture color. If the texture has an alpha component, the alpha value specifies the texture's transparency, allowing the geometry's color to show through the texture. In the above example, the lettering would be white and black.
ADD_TENV	adds the underlying geometry's color with the texture color.
DECAL_TENV	replaces the underlying geometry's color with the color of the texture. When this token is used with RGBA values, the alpha value determines the blending between the shape's and texture's color: when the alpha value is 1.0, the color is only the texture's; when the value is 0.0, the color is only that of the shape's.

Tip: If you use MODULATE, consider surrounding your texture images with a one-pixel border of white pixels and set **csTexture::setRepeatS()** and **csTexture::setRepeatT()** to CLAMP so the geometry's color is used where the texture runs out.

Color Components

A texture image can have up to four components per texture element:

- A *one-component image* consists of a luminance value, L_t. One-component textures are often referred to as intensity maps. For example, an image of a statue could use polygons of different intensities to shade and provide detail.
- A *two-component image* consists of luminance, L_t, and transparency, A_t. For example, you could create an architect's diagram of a house using polygons of different intensities to give detail to the building materials and then vary the transparency of the polygons to see through the building materials.
- A *three-component image* consists of a set of RGB values, referred to as a color triplet, C_t. For example, any color image is at least a three-component image.
- A *four-component image* consists of an RGB (or C_t) set of values, and transparency, A_t. The "t" subscript denotes the transparency or the color of the texture. For example, you could create an architect's diagram of a house using a variety of colors and transparencies.

The color components work with the texture environments in the following way:

- MODULATE works with any texture file.
- BLEND works with one- to four-component textures.
- REPLACE works with three- or four-component textures.
- ADD works with three- or four-component textures.
- DECAL works with three- or four-component textures.

Tip: MODULATE works best with bright materials because the texture intensity is reduced by the factor of the geometry's intensity.

Specifying Texture Coordinates

There are two ways to specify how a texture is applied to a geometry:

- Use the default. Cosmo 3D applies textures to geometries according to the geometry.
- Use the texture coordinate function, **setTexGen()**.

Using the Default

Cosmo 3D applies textures to geometries according to the geometry. For all geometries subclassed from **csGeometry**, Cosmo 3D

- Computes the bounding box.
- Turns the texture so its longest side is in the horizontal (s) direction.

The horizontal (s) value ranges from 0.0 to 1.0 and the vertical component ranges from 0.0 to *n*, where *n* equals the ratio of the t dimension to the s dimension; this ratio maintains the texture without distorting it.

Using the Texture Coordinate Function

The **setTexGen()** method generates texture coordinates by, in effect, projecting a texture plane onto a geometry, as shown in Figure 3-5.

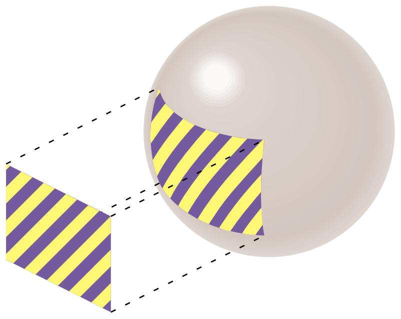

Figure 3-5 Texture Coordinate Function

The **setTexGen()** method specifies

- Whether or not the texture plane is repeated across the geometry.
- Whether the texture plane is stationary or moves in concert with the motion of the geometry.

The **setTexGen()** method takes a **csTexGen** object for an argument. In a **csTexGen** object, you set the

- Repetition of the texture image in three dimensions, s, t, and r.
- Mode of the texture in each of the dimensions.

For example, `csTexGen::setPlaneS(2.5, 0, 0, 0)` repeats the texture two-and-a-half times in the s dimension.

Figure 3-6 shows how a texture plane is repeated across a geometry.

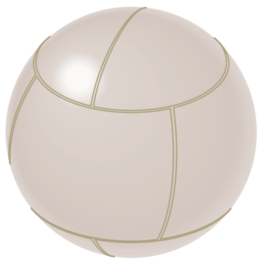

Figure 3-6 Repeated Texture on a Geometry

The default values of both s plane equations are (1,0,0,0), both t plane equations are (0,1,0,0), and all r and q plane equations are (0,0,0,0).

Setting the csTexGen Mode

If you think of the texture plane as being projected onto the surface of a geometry rather than being on the surface of a geometry, it is easy to understand how the mode settings in **csTexGen** work. Either the plane is stationary and the geometry moves “under” it or the plane moves in concert with the geometry. In the second case, the colors of the plane appear to be part of the geometry when it moves; in the first case, the colors of the plane appear to ride over the geometry when it moves.

You set the mode of each plane in **csTexGen** to one of the following values:

OFF	Turns off the texture.
EYE_LINEAR	Lets the geometry turn independently of the texture plane. In this case, the colors of the plane appear to ride over the geometry when it moves. This value is the default.
OBJECT_LINEAR	Lets the texture move in coordination with the geometry. In this case, the texture appears to be on the surface of the geometry.
SPHERE_MAP	Lets the texture pattern remain stationary as the geometry moves thus producing a mirror-like, circular reflection.
EYE_LINEAR_IDENT	Loads the identity matrix onto the modelview stack before loading the plane equation.

Enabling Texture Generation

The **setTexGen()** function is enabled or disabled using **setTexGenEnable()** with an argument of ON or OFF, respectively. Enabling the generation is, in effect, like turning on the light which shines through the plane and onto a geometry. Disabling the generation turns off the light.

The remaining sections apply to the appearance of the geometry itself.

Material Settings

The material field in **csAppearance** defines the surface qualities of a geometry, such as how well it reflects light, what color it reflects, and what color it emits. The material field is of type **csMaterial**, which has the following **set...()** methods:

```
void setAmbientIntensity(csFloat ambientintensity);
void setAmbientColor (const csVec3f& ambientColor);
void setDiffuseColor(const csVec3f& diffuseColor);
void setDiffuseColor(float v0, float v1, float v2);
void setSpecularColor(const csVec3f& specularColor);
void setSpecularColor(float v0, float v1, float v2);
void setEmissiveColor(const csVec3f& emissiveColor);
void setEmissiveColor(float v0, float v1, float v2);
void setAmbientIndex(csShort ambientIndex);
void setDiffuseIndex(csShort diffuseIndex);
void setSpecularIndex(csShort specularIndex);
void setShininess(csFloat shininess);
void setTransparency(csFloat transparency);
```

csMaterial also has a corresponding set of **get...()** methods.

Ambient color is the color of the light reflected from an object when lit by another ambient object in the scene. The default value is [0.2, 0.2, 0.2]. Ambient intensity refers to the strength of the reflection—a value between 0.0 and 1.0 where 1.0 is a strong reflection. Ambient index refers to a color lookup table in which each ambient color is paired with an index number for easy look-ups.

Diffuse color is an object’s base color. The default value is [0.8, 0.8, 0.8]. Diffuse index refers to a color lookup table in which each diffuse color is paired with an index number for easy look-ups.

Specular color is the reflected color of an object’s highlights. Specular intensity refers to the strength of the reflection. The default value is [0.0, 0.0, 0.0]. Specular index refers to a color lookup table in which each specular color is paired with an index number for easy look ups.

Emissive color is the color emitted by an object. A lamp shade for example, might have a base color of yellow. When the lamp is turned on, however, the emissive color might be white. The default value is [0.0, 0.0, 0.0].

Shininess describes how much of the surroundings are reflected by an object, for example, a mirror would have a large shininess value so that surrounding objects would be seen in it. Values range from 0.0, for a very dull surface, to 1.0, for a highly polished surface. The default value is 0.2.

Transparency describes how opaque or clear an object is, for example, water might be more clear than opaque. Values range from 0.0, for opaque, to 1.0, for complete transparency. The default value is 0.0.

Material Example

The following example shows the material settings for gold:

```
csMaterial *gold = new csMaterial;

gold->setAmbientColor(.3, .1, .1);
gold->setDiffuseColor(.8, .7, .2);
gold->setSpecularColor(.4, .3, .1);
gold->setShininess(.4);
```

Since gold is opaque, the default value, 0.0, for transparency suffices.

Filling Geometries

The **setPolyMode()** method specifies how to render the elementary polygons that compose a geometry. The following values for the method are valid:

POINT_PMODE
The polygon is rendered as points.

LINE_PMODE
The polygon is rendered as a line. This option is equivalent to rendering the geometry as a wireframe.

FILL_PMODE
The polygon is rendered as filled.

For example, if you use POINT_PMODE, a triangle would appear as three points. LINE_PMODE would render a triangle as a set of three lines; FILL_PMODE would render the triangle as filled.

Shade Model Settings

You set the shading model using the **setShadeModel()** method with one of the following **csContext::ShadeModelEnum** values as its argument:

FLAT_SHADE	Each primitive, geometric polygon that comprises a geometry has the same shade value. This option has the effect of making the primitive geometric polygons visible.
SMOOTH_SHADE	Shade values are interpolated across primitive geometric polygons. This option makes the primitive polygons look more like a curved surface.

Transparency Settings

To specify the transparency of a geometry locally, enable the transparency mode by setting **setTranspEnable()** to ON and then specifying the transparency mode in the argument of **setTranspMode()**. The possible transparency values include

FAST_TRANSP	Produces as quickly-rendered, lower-quality transparent geometry.
NICE_TRANSP	Produces a more slowly-rendered, higher-quality transparent geometry.
BLEND_TRANSP	Produces a smooth transparency between foreground and background images.
SCREEN_DOOR_TRANSP	Produces a mottled transparency, as though every other pixel is transparent.

Commonly, you use the **setTranspMode()** twice: you specify

- FAST_TRANSP or NICE_TRANSP
- BLEND_TRANSP or SCREEN_DOOR_TRANSP

Producing Transparency Without Blending

The **setAlphaFunc()** method sets the requirements for whether or not a pixel is rendered. Not rendering some of the pixels in a geometry has the effect of making the geometry partially transparent.

To use the **setAlphaFunc()** method, you first set the reference value against which you measure the alpha value of the pixel to be drawn using **setAlphaRef()**, for example

```
setAlphaRef(10);
```

Then you supply, as an argument to **setAlphaFunc(),** one of the values of **csContext::AlphaFuncEnum**:

- NEVER_AFUNC
- LESS_AFUNC
- EQUAL_AFUNC
- LEQUAL_AFUNC
- GREATER_AFUNC
- NOTEQUAL_AFUNC
- GEQUAL_AFUNC
- ALWAYS_AFUNC

For example, the following code

```
setAlphaRef(0.5);
setAlphaFunc(LESS_AFUNC);
```

is similar to the following lines of code:

```
if(Alpha < 0.5) {
    //draw the pixel };
```

where *Alpha* is the alpha values of pixels to be drawn in a geometry.

The values have the following effects:

NEVER_AFUNC	The incoming pixel is never rendered. This function has the effect of creating a totally transparent geometry.
LESS_AFUNC	The incoming pixel is rendered only if its alpha value is less than the reference value.
ALWAYS_AFUNC	The incoming pixel is always displayed. This function has the effect of creating an opaque geometry.

Chapter 4

Scene Graph Nodes

A *node* is an object that can be part of or entirely comprise a scene graph. Typically, a node is a collection of one or more fields and methods that together perform a specific function, for example, a **csShape** node encapsulates all information about the shape and appearance of a geometry.

Cosmo 3D nodes are divided into two types:

- group—associate other nodes.
- leaf— contain rendering information.

This chapter describes nodes and node types.

These are the sections in this chapter:

What Is a Node

A node is a collection of one or more fields and methods. Each field is a C++ class with data members and methods that get and set those member values. The fields set a variety of parameters. For example, some of the fields in the **csGeoSet** are summarized in Table 4-1.

Table 4-1 Examples of Fields in Nodes

Field Type	Fields	Description
SFRef	COORDS	Is a csCoordSet containing vertex coordinates.
SFRef	NORMALS	Is a csNormalSet containing normals for a geometry.
SFRef	COLORS	Is a csColorSet containing colors for a geometry.
SFRef	TEX_COORDS	Is a csTexCoord containing texture coordinates for a geometry.
SFRef	COORD_INDICES	Is a csIndexSet providing indices into a csCoordSet.
SFRef	NORMAL_INDICES	Is a csIndexSet providing indices into a csNormalSet.
SFRef	COLOR_INDICES	Is a csIndexSet providing indices into a csColorSet.
SFRef	TEX_COORD_INDICES	Is a csIndexSet providing indices into a csTexCoordSet.
SFEnum	CULL_FACE	Specifies whether to cull back-facing polygons, front-facing polygons, or no polygons.

Each node supplies default values for each of its fields.

Node Types

There are two types of nodes:

- Group—associates nodes into hierarchies.
- Leaf—sets the visual and audio values for a scene.

The following sections describe these node types.

Leaf Nodes

Leaf nodes are responsible for defining the visual and aural elements portrayed in a scene. Leaf nodes cannot have child nodes.

The following list shows all of the different types of Cosmo 3D leaf nodes; all are derivatives of **csLeaf.**

- **csShape—**associates a **csGeometry** object with a **csAppearance** object.
- **csLight—**is an abstract base class for light sources.
- **csDirectionalLight—**is a directional light source whose origin is at infinity.
- **csPointLight—**is a point source of light that radiates equally in all directions.
- **csSpotLight—**is a conical spot light.

The following section describes **csShape**. All of the other nodes are described in Chapter 8, "Lighting and Fog."

csShape

A **csShape** node associates a **csGeometry** object with a **csAppearance** object. Together, the **csGeometry** and **csAppearance** objects create a complete description of a shape.

To associate a **csGeometry** object with a **csAppearance** object, use the following methods:

```
void setAppearance(csAppearance* appearance);
void setGeometry(csGeometry* geometry);
```

There is a corresponding set of **get...()** methods that return the current appearance and geometry objects in the **csShape** object.

Group Nodes

Group nodes associate other nodes into a hierarchy known as a scene graph. Only group nodes can have children. A group node, for example, might associate two **csShape** nodes, as shown in Figure 4-1.

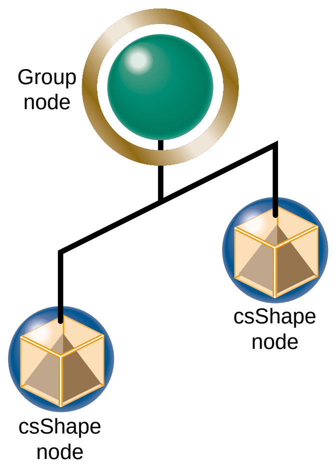

Figure 4-1 A Simple Grouping

Actions, such as a draw action, applied to a group node may be applied in no particular order to some or all of its children. A group node, then, defines the scope of an action.

For more information about actions, see Chapter 7, “Traversing the Scene Graph.”

Group Node Types

The following list shows all of the different types of Cosmo 3D group nodes; all are derivatives of **csGroup**:

- **csSwitch**—selects none, one, or all of its children, depending on its value. See “Using csSwitch to Switch Between Nodes” on page 59.
- **csBillboard**—rotates its children to face the viewer at the same time. See “Using csBillboard” on page 60.

- **csLOD—**(level-of-detail) is a switch that selects one of its children based on the distance between the camera and the shape encapsulated by the csLOD; the closer the shape, the greater the detail used when rendering the shape, the farther away the shape, the less detailed the shape. For more information, see Chapter 14, “Optimizing Rendering.”
- **csTransform**—positions and orients a shape in the coordinate system of the parent node to **csTransform**. For more information, see Chapter 6, “Placing Shapes in a Scene.”
- **csEnvironment—**is a grouping node which defines the scope of influence for the effects provided by **csLight**. For more information, see Chapter 8, “Lighting and Fog.”

Using csSwitch to Switch Between Nodes

The **csSwitch** node selects none, one, or all of its children. The constructor has the following prototype:

```
csSwitch();
```

To specify whether none, one, or all of a **csSwitch** node’s children are selected, use the following member function:

```
void setWhichChild(int which);
```

The possible values of *which* are

- NO_CHILDREN—to select no nodes
- an integer—to specify a child node
- ALL_CHILDREN—to select all of the children of the **csSwitch** node

Each child of a switch node is assigned an index number when added to the group node: the first child added is index 0, the second child added is index 1, and so on.

You might use a **csSwitch** node to create an animation sequence. For example, if each of the five child nodes of a **csSwitch** node contained the image of a character in different stages of walking, your application could switch sequentially between the child nodes to create a simple animation sequence.

Using csBillboard

csBillboard is a subclass of **csGroup**. It is used to rotate its children to face the viewer at all times. **csBillboard** has the following fields:

```
csMFRef children (inherited from csGroup)
csEnum mode
csMFVec3f position
csVec3f   axis
```

The mode field specifies one of three billboard modes: AXIAL, POINT_SCREEN, and POINT_OBJECT. The mode defines how the billboard's children should be rotated to face the viewer. Specific descriptions of each mode follow:

- AXIAL: In this mode, the local +z axis is rotated about the billboard axis to face the viewer (i.e. to match the negation of the view vector).
- POINT_SCREEN: In this mode, the local +z axis is rotated to face the viewer. The local +y axis is aligned with the screen's +y axis.
- POINT_OBJECT: In this mode, the local +z axis is rotated to face the viewer. The remaining degree of freedom is used to minimize the angle between the local +y axis and the billboard axis.

The position field specifies a position to which each child should be translated after it has been rotated to face the viewer. There should be as many entries in the position field as there are children.

The axis field is used in AXIAL and POINT_OBJECT modes.

Setting the Values in Scene Graph Nodes

Cosmo 3D allows you to set the values for nodes in two ways: either using the **set()** method in each of the node's fields, or by using tokens.

Setting the fields is different depending on whether or not the variable has a single or multiple value. If the variable has a single value, the variable can be set directly; if it has multiple values, the particular value in the set of values must be specified, as shown in Figure 4-2.

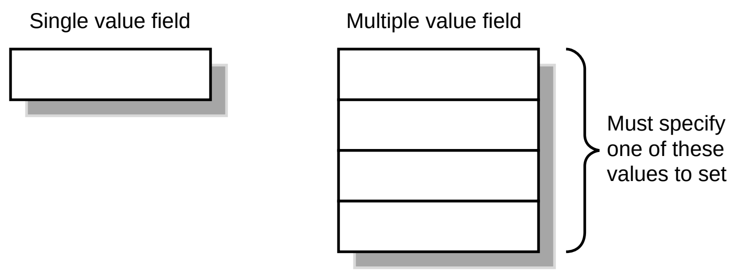

Figure 4-2 Setting Single and Multiple-Value Variables

Using set() and get() Methods to Set and Get Single-Value Fields

Nodes are composed of one or more fields, each of which is a class containing **set()**, **get()**, and, optionally, other methods. The **csAppearance** node, for example, contains many fields, some of which are **Shininess**, **Material**, and **TranspEnable** (enable transparency). To define one of these fields, you use the appropriate **set()** method, such as

```
csMaterial *mtl->setShininess(ShininessValue);

ShininessValue = mtl->getShininess();
color = mtl->getDiffuseColor();
```

ShininessValue is a float. The first line of code sets the shininess value of the **csMaterial**, *mtl*. The following lines return an atomic, single value, *ShininessValue*, and a composite, single value, *color*. A composite, single value is a set of numbers that represent a single feature, for example, RGB values represent one feature: color

Using Tokens to Set and Get Single-Value Fields

To use tokens to set or get single-value fields, you

1. Get a handle to the field specified by the token.
2. Use the handle to set or get the field.

For example:

```
F = mtl->getField(SHININESS);
F->set(ShininessValue);

ShininessValue = F->get();
F->get(c);
```

ShininessValue is a float. The first line of code returns a handle, *F*, to the shininess field. The second line then sets the value of that field.

The third line returns an atomic, single value. Since there is only one value, it does not need to be specified in the argument. The last line returns a composite, single value, *F*.

Using set() and get() Methods to Set and Get Multiple-Value Fields

Multiple-value fields are arrays of variables. For example, the **csMaterial** field has a number of values, including

```
sfFloat Shininess;
sfVec DiffuseColor;
```

To get or set a value in a **csMaterial** field, you must specify which of the values in the field you are retrieving or setting, for example:

```
csGroup *g;
g->addChild(child);

child = g->getChild(1);
```

Child nodes of a group node are numbered, starting with zero. To retrieve the specific child in the **csGroup**, you must specify in the argument of **getChild()** which child you want returned; in this example, it is child number one.

Using Tokens to Set and Get Multiple-Value Fields

To use tokens to set or get multiple-value fields, you

1. Get a handle to the field specified by the token.
2. Use the handle to set or get values from a specific variable in the field.

For example:

```
csMFRef *F;
csGroup *g;

F = g->getField(CHILDREN);

F->append(child);
child = F->get(1);
```

To retrieve the specific child in the **csGroup**, the last line of code shows that you must specify in the argument of **get()** which child you want returned; in this example, it is child number one.

Chapter 5

Building a Scene Graph

A scene graph can be a single node or a hierarchy of nodes, as shown in Figure 5-1.

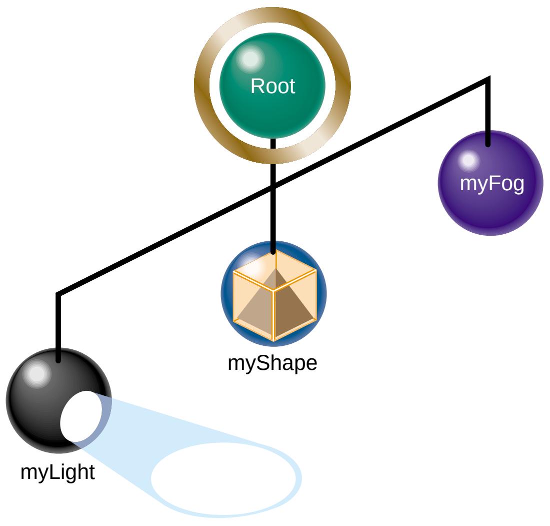

Figure 5-1 Scene Graph

The hierarchy specifies the order in which the nodes are acted upon when an action is applied to the scene graph. The hierarchy is established by the order in which the group nodes are added to the branches in a scene graph branch.

This chapter describes how to build and edit a scene graph.

This chapter includes the following sections:

Creating Scene Graphs

The top node in a scene graph is called the root node; it must be a group-type node. Actions applied to the root node visit all of the children nodes of the root node.

To create a scene graph, you start with the root node and add children to it using the **csGroup::addChild()** method, as follows:

```
root->addChild(myLight);
root->addChild(myShape);
```

In this simple example, the *myLight* node is added first to the scene graph whose root node is called *root*; the *myShape* node is added second. When a draw action is applied to the root node, either of these nodes may be evaluated first.

To complete the scene graph, you add children to any child nodes of the root node that are a group-type. You continue adding children to group-type nodes until the complete scene is encapsulated in the scene graph.

Root Node

Most scene graphs have a root node of type **csGroup**. If you have a one-node scene graph, for example, a **csShape** node, which is a leaf node, then the root node, the only node, is a leaf node.

It is also possible to have multiple root nodes for a scene graph, as shown in Figure 5-2.

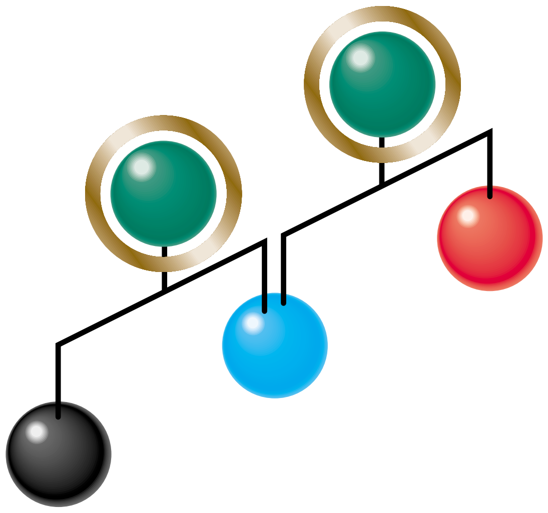

Figure 5-2 Multiple Root Nodes

Applying Actions to Multiple Root Nodes

If a scene graph has multiple root nodes, an action applied to one of the roots would only traverse the descendants of the specific root node. While this hierarchy of nodes is legal, if your application is going to draw both scene graphs anyway, you should create a single root node common to both scene graphs. In Figure 5-2, this could be done by adding a group node above the root nodes shown, thus making them children of the single group node. The advantage to this construction is that you do not have to apply the same action repeatedly to different root nodes.

Actions applied to a root node flow (potentially) to all of the other nodes in the scene graph. Passing the action from one node to another is called *traversing*. As an action traverses a scene graph, variables set by the nodes in the scene graph change the graphical context, which, in turn, changes the objects in the scene according to the node values.

Creating A Sample Scene Graph

Example 5-1 is a simple scene graph.

Example 5-1 A Simple Scene Graph

```
// create the root node of the scene graph
csGroup *root = new csGroup;

// create the nodes for the scene graph
csSpotLight *mySpotLight = new csSpotLight;
csShape *myShape1 = new csShape;
csShape *myShape2 = new csShape;


// Add the nodes to the group node to create a scene graph
root->addChild(mySpotLight);
root->addChild(myShape1);
root->addChild(myShape2);
```

In this example, a root node is created using the **new** directive and children nodes are added to it using the **csGroup::addChild()** method. The order in which the children are added to the root node is the order in which the nodes are acted upon when an action is applied to the root node.

Diagramming Scene Graphs

Diagramming a scene graph is helpful in visualizing the structure of a Cosmo 3D application. Figure 5-3 shows a diagram representing the scene graph coded in Example 5-1.

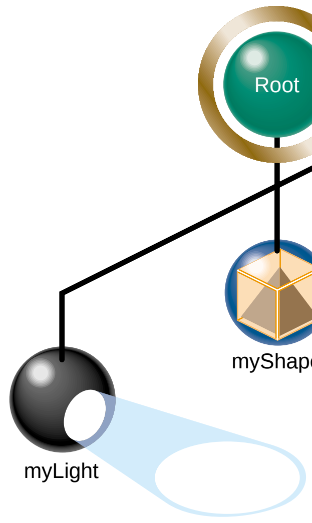

Figure 5-3 Simple Scene Graph

In diagrams of scene graphs, circles represent nodes and lines represent the node hierarchy. The different types of circles represent the different types of nodes, for example, *root* is a **csGroup**-type node whereas **myShape** is a **csLeaf**-type node. Notice how the nodes are positioned: the three leaf nodes are children of the **csGroup** node and the leaf nodes appear in the same order in which they were added to the root node in Example 5-1. Remember, however, that actions may traverse these leaf nodes in any order because these leaf nodes are at the same level.

Scene Graph Diagrams At A Glance

Diagrams of scene graphs provide an overview of the functionality of a Cosmo 3D application without the bother of delving into the complexity of the code. A diagram of a scene graph, for example, can show the number of data sets that can be rendered.

For more information about the order in which nodes are acted upon, see "The Order In Which Actions Are Passed Between Nodes" on page 86.

There are no rules for constructing a scene graph, however, it is customary to organize it in the following way:

- Reading the nodes left to right shows you the different geometries rendered in a scene graph.
- Reading the nodes from top to bottom shows you the different parts that are combined to form a larger geometry.

For example, reading horizontally, Figure 5-4 shows that the two subgraphs, Molecules and Hydrogen bonds, are separate geometries that appear together in world space.

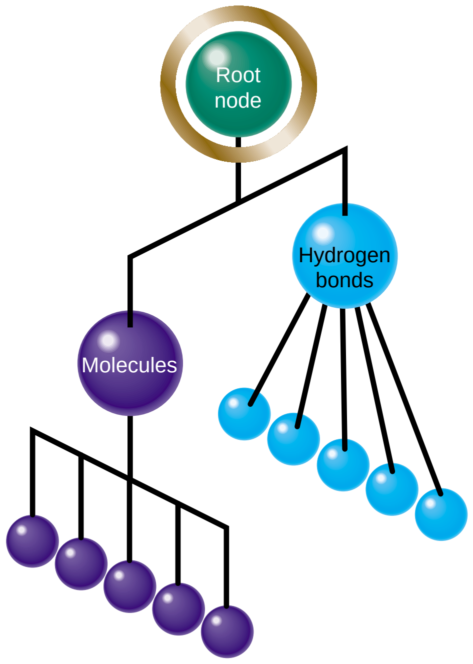

Figure 5-4 Two Sets of Data Rendered Differently

In the subgraph shown in Figure 5-5, you can see that the foot, leg, and torso nodes are parts which, when rendered together, display the lower half of a body.

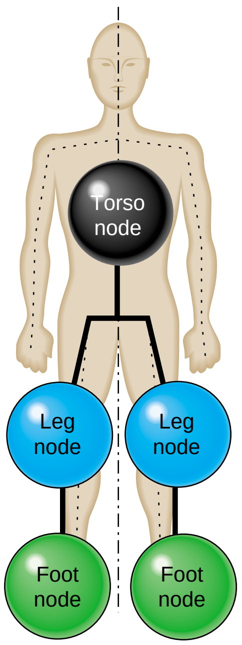

Figure 5-5 Torso Subgraph

Because the left leg looks different from the right leg, you need two different shape nodes. If, however, you want to display the same geometry twice, but in different locations, you can use two transformation nodes to place the same object in different locations, as shown in Figure 5-6.

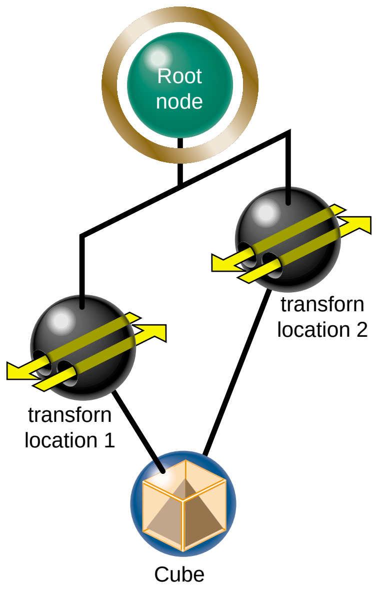

Figure 5-6 Showing the Same Geometry in Two Locations

In this example, the scene graph makes it easy to see that a cube is rendered in two locations by two **csTransform** nodes.

Altering Scene Graphs

After using **csGroup::addChild()** to create a scene graph, you can use the following methods to edit it:

```
void removeChild(int i);
int  removeChild(csNode *node);
int  replaceChild(csNode *old, csNode *node);
void insertChild(int i, csNode *node);
```

These methods allow you to remove, replace, or insert a child node, respectively. For example, to insert a node between two children, use the **insertChild()** method:

```
csShape *myShape = new csShape;
root->insertChild(2, myShape);
```

The children nodes are numbered starting with 0. The "2" in the argument of **insertChild()** specifies that the *my*Shape node should be inserted in the scene graph as the number two node.

Note: Although leaf nodes attached to the same group node can be acted upon in any order, the first node added to a group node is always node zero, the second node added to the root node in the code is node one, and so forth.

csGroup also supplies the following methods for finding the number of a node in a scene graph, returning the number of children in a scene graph, and setting the number of nodes in a group, respectively.

```
int  searchChild(csNode *node);
int  getChildCount();
void setChildCount(int count);
```

In general, you use the **searchChild()** method to return the number of a node so you can perform other functions on or around it, such as replacing it.

Loading a VRML Scene Graph

Example 5-2 shows a portion of *vrml.cxx*. The example illustrates how to load a VRML scene graph using **vlDB::readFile()**.

Example 5-2 Loading a VRML Scene Graph

```
csGroup             *vrml = new csGroup;

    for (int i=1; i<argc; i++)
    {
        csContainer             *v;
        static char             path[512];
        char                    *lastSlash;

        strcpy(path, argv[i]);
        lastSlash = strrchr(path, ‘/’);
        if (lastSlash != NULL)
            *lastSlash = ‘\0’;

        strcat(path, “:.”);
        csGlobal::setFilePath(path);

        if (vlDB::readFile(argv[i], v, viewPoints) && v != NULL)
        {
            printf(“Read %s was ok\n”, argv[i]);
            vrml->addChild((csNode*)v);
        }
        else
            printf(“Read %s was bad\n”, argv[i]);
    }

    new csWindow(“vrml”);
```

Saving Scene Graphs

The data in the scene graph database is not necessarily static. You might, therefore, need to save scene graph data into a file. To do so, you use the following method:

```
csGlobal::storeFile(NameOfFile, *dataStructure);
```

where *NameOfFile* is the name of the file where you want to store the data and *dataStructure* is the Cosmo 3D in-memory data structure to store. The method returns TRUE if the file is stored successfully, FALSE otherwise.

Note: **storeFile()** is used with Cosmo 3D-only applications. If you are writing an Optimizer application, do not use **storeFile()**.

Troubleshooting Scene Graph Construction

A common mistake in Cosmo 3D applications is forgetting to include a **csLight** or **csCamera** node. The **csDrawAction::setCamera()** method specifies the camera and points it at the shapes in the scene graph.

For more information about **csLight** or **csCamera**, see Chapter 8, “Lighting and Fog” and Chapter 9, “Viewing the Scene,” respectively.

Another common error in Cosmo 3D applications is pointing the camera in the wrong direction in which case the camera may produce a blank image.

Chapter 6

Placing Shapes in a Scene

When you create a geometry, it has a specified size, location, and orientation, as defined in its own space. You place such a geometry

- In relationship to other shapes in the same scene.
- Into the coordinate system of the root node, known as world space.

This chapter describes how to perform each of those tasks.

The final transformation that affects the view of the user is that created with the camera. Rotating the camera has an obvious affect on the view of the scene. To read more about the camera transformation, see “Using a Camera to View a Scene” on page 98.

This chapter has the following sections:

- “Creating a Sense of Depth” on page 77.
- “Transforming Shapes to New Locations, Sizes, and Orientations” on page 79.

Creating a Sense of Depth

Geometries are layered by Cosmo 3D according to the order in which they are rendered. For example, the first geometry rendered is usually covered by the second geometry if they overlap.

Overriding the Default Order of Layering Shapes

To override this layering effect, you can use the **csContext::setcsDepthFunc()** method; it determines the layering order of geometries in a scene according to values in the Z dimension. To specify a layering method, use one of the tokens in **csContext::DepthFuncEnum**.

NEVER_DFUNC

LESS_DFUNC

EQUAL_DFUNC

LEQUAL_DFUNC

GREATER_DFUNC

NOTEQUAL_DFUNC

GEQUAL_DFUNC

ALWAYS_DFUNC

Here are the effects of some of the arguments:

NEVER_DFUNC	the incoming pixel is never displayed on top of the current, corresponding pixel in the buffer. This function has the effect of reversing the normal order of layering: pixels are rendered behind the pixels currently in the buffer.
LESS_DFUNC	A pixel is displayed only if its Z component is less than the Z value of the corresponding pixel currently in the buffer. This function presents the intuitive representation of close objects appearing in front of distant objects.
ALWAYS_DFUNC	The incoming pixel is always displayed on top of what is currently displayed regardless of the Z component.

The default is LEQUAL_DFUNC.

Transforming Shapes to New Locations, Sizes, and Orientations

The **csTransform** node

- Allows you to specify vertex coordinates of a shape in local space, using (0, 0, 0) as the origin.
- Translates the local coordinates into the coordinates of its parent node.

If there is more than one **csTransform** node in a hierarchy of nodes, each **csTransform** node translates the child node coordinates into coordinates of its parents all the way up the hierarchy until a final **csTransform** node translates the coordinates of the shape into those of the root node. The coordinate system of the root node is called *world space* because all shapes in all parts of the scene graph are translated into that coordinate system.

Placing Transform Nodes

Typically, **csTransform** nodes are placed between **csShape** or **csGroup**-type nodes and the rest of the scene graph, as shown in Figure 6-1.

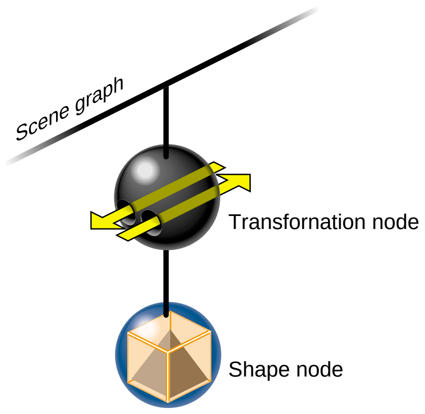

Figure 6-1 Placement of csTransform Nodes

Any node, however, can be the child of a transformation node.

Setting the Transformation

The **csTransform** node allows you to set the location (translation), rotation, and scale of its children using the following methods:

```
void setTranslation(const csVec3f& translation);
void setTranslation(csFloat v0, csFloat v1, csFloat v2);

void setRotation(const csRotation& rotation);
void setRotation(csFloat v0, csFloat v1, csFloat v2, csFloat v3);

void setScale(const csVec3f& scale);
void setScale(csFloat v0, csFloat v1, csFloat v2);

void setScaleOrientation(const csRotation& scaleOrientation);
void setScaleOrientation(csFloat v0, csFloat v1, csFloat v2,
    csFloat v3);

void setCenter(const csVec3f& center);
void setCenter(csFloat v0, csFloat v1, csFloat v2);
void setMatrix(Matrix4f mat)
```

There is a corresponding set of **get...()** methods for each of these methods.

Each method is overridden so that you can specify the arguments to the methods either as objects or individual coordinates.

setTranslation() positions the children of the transform in the space of the node that is the parent to **csTransform**.

setCenter() specifies the point around which an object rotates.

setRotation() rotates the children of the transform around the point specified in **setCenter()**.

setScale() specify the scale factor of the children of the transform along the X, Y, and Z axes.

setScaleOrientation() specifies the orientation in which the scaling takes effect. Figure 6-2 shows a shape scaled by a factor of 2 in two different orientations, 0 and 45 degrees, respectively.

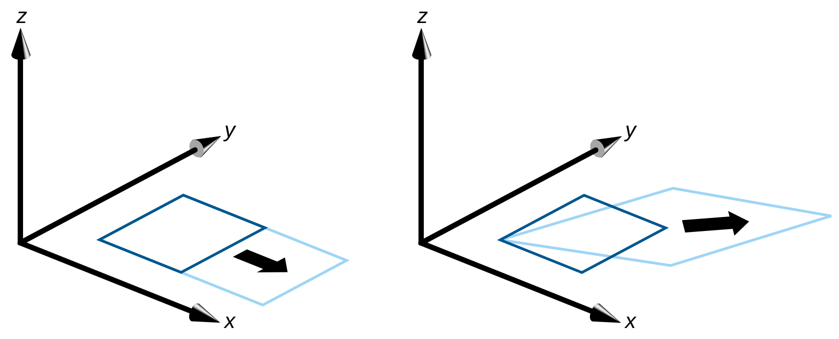

Figure 6-2 Scaling in Different Orientations

All of these methods invisibly set a transformation matrix to carry out their actions. If you want to set the matrix directly, you can use the **setMatrix()** method.

Ordering Transformations

The order in which you perform transformations can effect the final result. Take, for example, translating and rotating a model. If you perform the transformations in this order, you end up with a rotated model translated, for example, down the X axis, as shown in Figure 6-3.

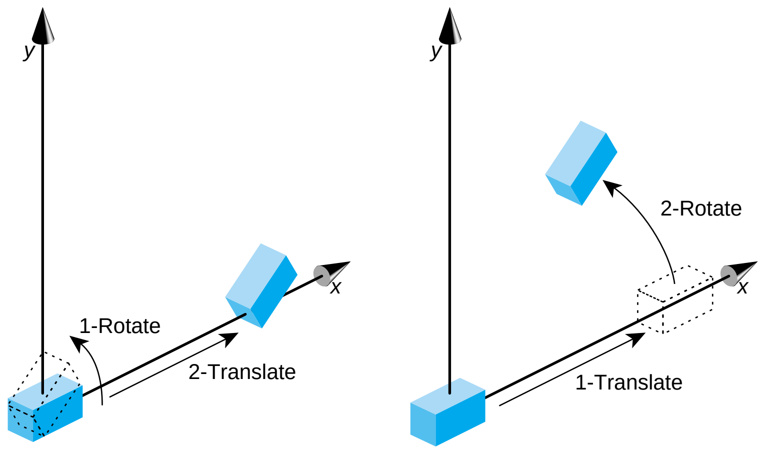

Figure 6-3 Order of Transformations

When you reverse the order of the transformations, the end result is different. Since the center of rotation is the origin, the rotation transformation lifts the object above the X axis.

Placing Geometries in World Space

Multiple transformation nodes can orient and size all shapes in a scene graph into the space of the root node. The space of the root node is called world space.

World space is the coordinate system of the root node in which all shapes in a scene graph can reside. *Local space* is a coordinate system in a subsection of a scene graph.

Cosmo 3D Matrices

Many geometry variables are defined in local space. To translate those values into the world space you use a transformation matrix. The transformation matrix is a 4 × 4 matrix of type **csMatrix4f** that contains scaling, translation, and rotation information. You can set the matrix explicitly or, more easily, you can use class methods to generate a transformation matrix.

Cosmo 3D matrices are column major, which means their members are ordered in the following way:

```
0  1  2  3
4  5  6  7
8  9  10 11
12 13 14 15
```

You can set a transform matrix directly, or you can use **csContext** methods to scale and orient a shape in world space.

Chapter 7

Traversing the Scene Graph

Once you create a scene graph, you apply an action to the root node to trigger the events prescribed in the scene graph nodes. Most nodes include an overwritten **apply()** method that triggers an appropriate response when a specific action is applied to the node. Actions include rendering the scene (draw action) and playing sound files (sound action).

This chapter describes how an action traverses a scene graph and the actions available in Cosmo 3D.

This chapter has the following sections:

- "Scene Graph Actions" on page 83.
- "The Order In Which Actions Are Passed Between Nodes" on page 86.

Scene Graph Actions

An action is an operation that is carried out on one or more nodes in a scene graph. In response to actions,

- **csLeaf** nodes set values.
- **csGroup** nodes pass actions from one child node to another.

For example, when a **csDrawAction** is applied to the root node of a scene graph, the action operates on (potentially) all of the nodes in the scene graph so that all of the **csShape** nodes in it are rendered.

The action-specific responses taken by a node are implemented in the following node functions:

- draw()—for **csDrawAction**
- isect()—for **csIsectAction**
- compile()—for **csCompileAction**
- sound()—for **csSoundAction**

When an action operates on one node after another, the action is said to be *traversing* the scene graph.

Action Types

In Cosmo 3D, there are four kinds of actions:

- **csDrawAction**—Renders a scene graph.
- **csIsectAction**—Selects objects intersecting a ray.
- **csCompileAction**—Compiles a subgraph.
- **csSoundAction**—Plays spatialized (3D) sounds in a scene graph.

All of the actions are derived from **csAction**.

For more information about

- **csIsectAction**, see "Using csIsectAction" on page 140.
- **csCompileAction**, see "Compiling Part of a Scene Graph" on page 164.

csAction

csAction is a virtual class from which all actions are derived. An action is an operation that is performed on some or all of the nodes in a scene graph, for example, a **csDrawAction**, when applied to the root node, renders a scene graph.

csAction contains the following method:

```
virtual void  apply(csNode *node) = 0;
```

You invoke an action by creating an instance of an action class and applying it to a node (commonly the root node), for example:

```
csGroup* rootNode = new csGroup;
...
csDrawAction* renderAction = new csDrawAction;
renderAction->apply(rootNode);
```

In this example, a draw action, *renderAction*, is applied to the root node, *rootNode*, of a scene graph.

When an action is applied to a node, each kind of node responds in its own way. When a draw action is invoked on a leaf node, for example, it sets the values that describe a shape. When a draw action is invoked on a group node, it applies the action to some or all of its children.

Rendering the Scene

A **csDrawAction** causes leaf nodes to set the variables that affect the rendering of the shapes in a scene graph and causes the shapes to be rendered.

A **csDrawAction** has the following get and set methods:

```
void reset();
virtual csTravDirective apply(csNode *node);
```

reset() resets the cull stack and transformation stack.

The virtual function **apply()** is the method you use to apply the draw action on a node in a scene graph, for example:

```
renderAction->apply(rootNode);
```

where *renderAction* is a **csDrawAction** instance and *rootNode* is the root node of a scene graph. A draw action can be applied to any node in a scene graph. If you apply a draw action to a leaf node, the node may set a value but the action does not spread to any other nodes.

Playing Sound Files

csSoundAction plays sound files. The **csSound** node indirectly specifies—through **csAudioClip** and **csAudioSamples** nodes—the sound file to play. The **csSoundAction** places the listener (referred to as the microphone in Cosmo 3D) relative to the sound source for spatial effects, such as volume based on the proximity of the listener to the sound source.

```
void        setMicrophone(Microphone mic);
Microphone  getMicrophone();
```

When the **csSoundAction** traverses a scene graph, it creates a list of active sounds in the scene graph and supplies that information to **csContext** internally. If any sounds are active, **csContext** sends instructions to play the associated sound files for a specific duration.

For more information about sounds, see Chapter 15, "Adding Sounds To Virtual Worlds."

The Order In Which Actions Are Passed Between Nodes

In a scene graph, group nodes are acted upon in a top-to-bottom sequence. Leaf nodes under any one group node are acted upon in an unspecified order. While not every child node may be visited, for example, under a **csSwitch** node, it is guaranteed that all parent nodes are visited before their children nodes.

Top-Down Traversals

In diagrams of scene graphs, this order of node evaluation, called an in-order traversal, is represented by the layout of the nodes so that actions traverse from the top to bottom of the scene graph. For example, the numbers in Figure 7-1 show one order in which an action might traverse the nodes in a scene graph.

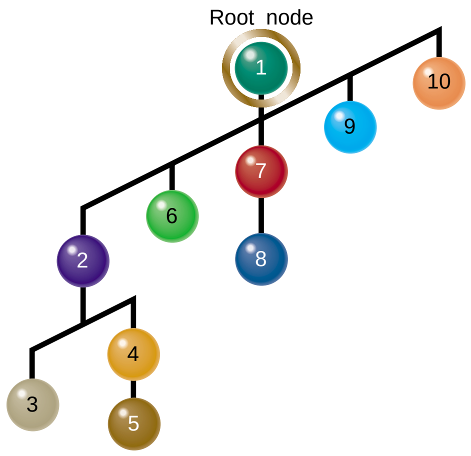

Figure 7-1 The Flow of an Action Through A Scene Graph

A top-down traversal means that a node can never affect a node “above” it. For example, node 3 in Figure 7-1 cannot affect nodes 4 and 5, however, node 4 can affect node 5. So, that branch of the scene graph is traversed in the following way when an action is applied to the root node:

1. The action traverses from node 1 to any of its children.
2. When the action visits node 2, node 2 propagates the action to one of its children.
3. If the action propagates to node 3, the action takes effect and the attribute variables are reset to their values prior to node 3 after which the action visits node four.
4. The action then traverses nodes 4 and 5 at which point the traversal state is reset to its state prior to node 2.
5. The action then traverses to any of the other child nodes of node 1.

Each node can immediately change the image in the frame buffer according to the content to the node.

Whether all of the children of a group node are evaluated depends on the group node type. For example, in Figure 7-1, if the number 1 node is of type **csSwitch**, the traversal may visit all, one, or none of nodes 2, 6, 7, 9, and 10.

Chapter 8

Lighting and Fog

This chapter discusses two features implemented by subclasses of **csEnvironment**, lighting and fog.

- Lights illuminate shapes in a scene. Without lights, shapes are not visible.

  To limit the range of a light, such as limiting the rays of a light to the room it is in, you include the lights in the light array in **csEnvironment**.
- Fog makes an image appear more natural by fading objects in the distance. You include fog in **csEnvironment**.

This chapter describes first how to use lights, change the shadow modeling, and change the screen to one color. It then discusses how you can use fog for atmospheric effects such as smoke, haze, or mist.

This chapter contains the following sections:

- "Using Lights in Scenes" on page 89.
- "Limiting the Scope of Lights" on page 92.
- "Using Fog in Scenes" on page 93.

Using Lights in Scenes

Cosmo 3D provides a variety of light types. This chapter describes the light types presented in Cosmo 3D in addition to the virtual light class, **csLight**, from which you can create your own light objects.

csLight

csLight is an abstract base class for light sources. It provides the following methods for setting light values:

```
void setOn(csBool on);
void setIntensity(csFloat intensity);
void setAmbientIntensity(csFloat ambientIntensity);
void setColor(const csVec3f& color);
void setColor(csFloat v0, csFloat v1, csFloat v2);
```

There is a corresponding set of **get...()** methods that return each of the light settings.

setOn() turns on the **csLight** object.

setIntensity() sets the brightness of the **csLight** object.

setAmbientIntensity() sets the brightness of the ambient color. The ambient intensity is how bright the light makes all objects appear.

setColor() sets the color of the **csLight** object.

csDirectionalLight

csDirectionalLight is a directional light source that approximates distant light sources, such as the sun, and can improve rendering performance over local light sources, such as **csPointLight** and **csSpotLight**. Use **csDirectionalLight** to set the direction of general lighting for a scene using one of the following methods:

```
void setDirection(const csVec3f& direction);
void setDirection(csFloat v0, csFloat v1, csFloat v2);
```

There is a corresponding set of **get...()** methods that return the lighting direction.

A **csDirectionalLight** has no range limitations so it affects all children of a **csEnvironment** object when included in a light array.

Note: A **csDirectionalLight** object's direction is affected by all **csTransform** nodes above it in the scene hierarchy.

csSpotLight

csSpotLight is a directional light source. Because **csSpotLight** is subclassed from **csPointLight**, **csSpotLight** can be positioned.

The light emanates as a cone; the axis of the cone specifies the direction of the spot light and is defined in the following methods:

```
void setDirection (const csVec3f& direction)
void setDirection (csFloat v0, csFloat v1, csFloat v2)
```

The intensity distribution and the cut off angle of the light are set with the following methods:

```
void setExponent(csFloat component);
void setCutOffAngle(csFloat cutOffAngle);
```

csPointLight

csPointLight is a point light source that radiates equally in all directions. The range of a **csPointLight**'s effect is localized to a **csEnvironment** object when the **csPointLight** is included in its light array.

All descendants of a **csEnvironment** object that lie within the **csPointLight**'s shining radius are affected by the **csPointLight**. **csTransform** objects affect the location and shining radius of each **csPointLight**.

Use the following methods to define a **csPointLight.**

```
void        setLocation(const csVec3f& location);
void        setLocation(csFloat v0, csFloat v1, csFloat v2);

void        setRadius(csFloat radius);

void        setAttenuation(const csVec3f& attenuation);
void        setAttenuation(csFloat v0, csFloat v1, csFloat v2);
```

There is a corresponding set of **get...()** methods that return the current point light settings.

setLocation() defines the location of the **csPointLight**.

setRadius() defines the maximum range of the light.

setAttenuation() defines how quickly the intensity of the light declines over distance.

Limiting the Scope of Lights

csEnvironment defines the scope of environmental effects, such as how far light from a **csLight** object can travel. When you create a virtual room, the goal is to make a lamp in the room shine in the room only—not leak through walls into the hallway. When you make a **csLight** part of the light array in **csEnvironment**, the lamp light stops at the walls of the room.

Another application of **csEnvironment** is rendering headlights on a car. The goal is to have the lights move with the car and extend only a couple of hundred feet in front of the car. To do that, you add a **csPointLight** to the **csEnvironment** light array and limit the **csPointLight** to several hundred feet.

The Scope of the Light Array

The **csEnvironment** node serves as the root node for the effects of all lights in its array. **csEnvironment** uses an array of lights because you might have more than one **csLight** in a room, but the light from all of the lamps should end at the walls of the room. All **csLight**s in the light array have the same range limitations.

csEnvironment Methods

csEnvironment contains the following method that specifies an array of lights:

```
csMFRef* light() const;
```

For example, if you wanted to add two lights but remove a third, you would use code similar to the following:

```
// create an Environment
csEnvironment* park = new csEnvironment;
...

// find and remove light #3
csLight* badLight = park->getLight(3);
park->light()->remove(badLight);

//create two lights
csSpotLight* spot = new csSpotLight;
csPointLight* flood = new csPointLight;
```

```
// add the lights to the environment light array
park->light()->append(3, spot);
park->light()->append(4, flood);
```

Using Fog in Scenes

Fog is generated by setting up a **csFog** node and adding it to the **csEnvironment**. This fog description will affect all children of the **csEnvironment**. Subsequent **csFog** nodes will override the current fog description. Fog should also be globally enabled via **csContext::setFogEnable()** and can be overridden with **csAppearance::setFogEnable()**. If **csContext::FogEnable** is FALSE, it is disabled for the entire scene graph (unless overridden by **csAppearance::fogEnable** regardless of the value of **csFog::on**.

This section discusses the following fog-related topics:

- "Uses of Fog in Cosmo 3D Applications" explores the different uses for fog and reasons for using it.
- "How to Use Fog in Cosmo 3D Applications" explains how to use fog in a Cosmo 3D application.
- "How to Use Fog" is a code fragment illustrating the information from the preceding section.

Uses of Fog in Cosmo 3D Applications

Computer images sometimes seem unrealistically sharp and well defined. You can make an entire image appear more natural by adding fog, which makes objects fade into the distance. Fog is a general term that describes similar forms of atmospheric effects; it can be used to simulate haze, mist, smoke, or pollution. Fog is essential in visual-simulation applications, where limited visibility needs to be approximated. It is often incorporated into flight-simulator displays.

When fog is enabled, objects that are farther from the viewpoint begin to fade into the fog color. You can control the density of the fog, which determines the rate at which objects fade as the distance increases, as well as the fog's color. Since fog is applied after matrix transformations, lighting, and texturing are performed, it affects transformed, lit, and textured objects. Note that with large simulation programs, fog can improve performance because you can choose not to draw objects that would be too fogged to be visible.

All types of geometric primitives can be fogged, including points and lines. Using the fog effect on points and lines is also called depth-cuing and is popular in molecular modeling and other applications.

How to Use Fog in Cosmo 3D Applications

Fog is generated by setting up a **csFog** node and adding it to the **csEnvironment**. The **csFog** class has the following fields, which determine its effects:

Table 8-1 Fields in **csFog**

Type	Name	Enumerant	Default
csSFBool	on	ON	false
csSFEnum	mode	MODE	FOG_EXP
csSFFloat	density	FOG_DENSITY	1.0
csSFFloat	start	FOG_START	0.0
csSFFloat	end	FOG_END	1.0
csSFFloat	index	FOG_INDEX	0.0
csSFVec4f	color	FOG_COLOR	{0.0, 0.0, 0.0, 0.0}

The effects of the different fields are parallel to those of OpenGL **glFog()**, and can be examined in the *OpenGL Reference Manual* or the glFog manpage.

Enabling Fog

There are three different fields that determine whether fog is enabled or not:

- **csContext::fogEnable**
- **csAppearance::fogEnable**
- **csFog::on**

In general whoever is OFF has precedence.

- If **csContext::fogEnable** is OFF, no fog is drawn unless there is a **csAppearance** that has **fogEnable** ON. In this case, fog is enabled for that shape *only* if it is within the scope of a fog node that is turned on (that is, under a **csEnvironment** that has fog enabled).
- If the **csFog** node is OFF, there will be no fog drawn in any circumstances, even if the fogEnable in the **csAppearance** or **csContext** is TRUE.

The fog description will affect all children of the **csEnvironment**. Subsequent **csFog** nodes override the current fog description.

How to Use Fog

The following code fragments from the **csFog** manpage illustrates how to use of fog:

```
static csVec4f fogClr(0.2f, 0.2f, 0.2f, 1.0f);

     <...>

     // Create environment node for scene
     csEnvironment *env = new csEnvironment;

     // Add enabled fog node to environment
     csFog* fog = new csFog;
     fog->setOn(TRUE);
     fog->setColor(fogClr);
     fog->setEnd(400.0f);
     fog->setMode(csFog::LINEAR_FOG);
     env->setFog(fog);

     // Enable fog for default state
     ctx->setFogEnable(TRUE);

     <...>

     // In frame function
     ctx->clear(csContext::COLOR_CLEAR | csContext::DEPTH_CLEAR,
                fogClr[0], fogClr[1], fogClr[2], fogClr[3]);
```

Chapter 9

Viewing the Scene

To view a scene, you must define:

- The size of the viewport.
- The position and the orientation of the camera.

This chapter describes how to set up the viewport and how to use cameras to view a scene.

This chapter has the following sections:

- "Setting the Screen Display of the Scene" on page 97.
- "Using a Camera to View a Scene" on page 98.

Setting the Screen Display of the Scene

The viewport is the rectangular portal through which you view a scene. The viewport can be as large as a **csWindow**, or it can be just a portion of a **csWindow**, as shown in Figure 9-1.

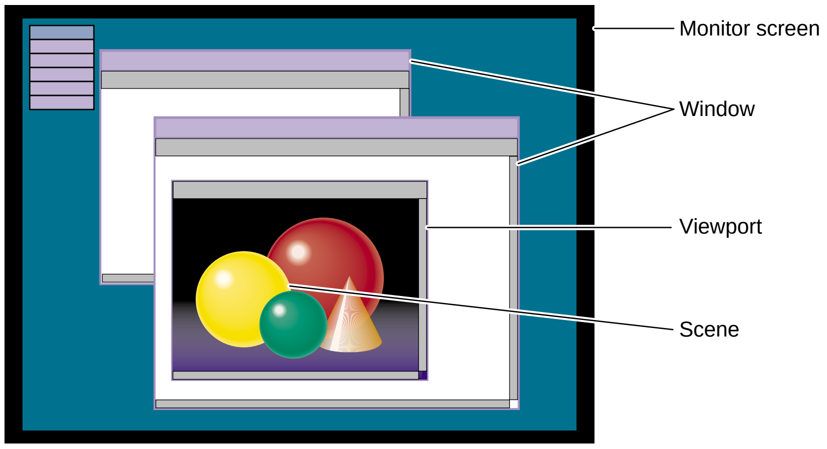

Figure 9-1 Viewport

You set the size of the viewport using the following **csContext** method:

```
static void setViewport(csInt x, csInt y, csInt w, csInt h);
```

x and *y* are the coordinates of the lower, left corner of the viewport (where the lower, left corner of the **csWindow** is (0. 0)).

w and *h* are the width and height of the viewport.

Using a Camera to View a Scene

Cosmo 3D provides a variety of cameras to position, orient, and delimit the view of the shapes in the scene graph. This section describes the different cameras that are available, including:

- “csCamera” on page 99.
- “csOrthoCamera” on page 101.
- “csPerspCamera” on page 101.
- “csFrustumCamera” on page 105.

csCamera

csCamera is an abstract base class from which all other cameras are derived. **csCamera** defines the viewing volume. The viewing volume is bounded by the camera's origin and orientation, the far clip plane, and an aspect ratio. The aspect ratio is defined as the image's width divided by its height, as shown in Figure 9-2.

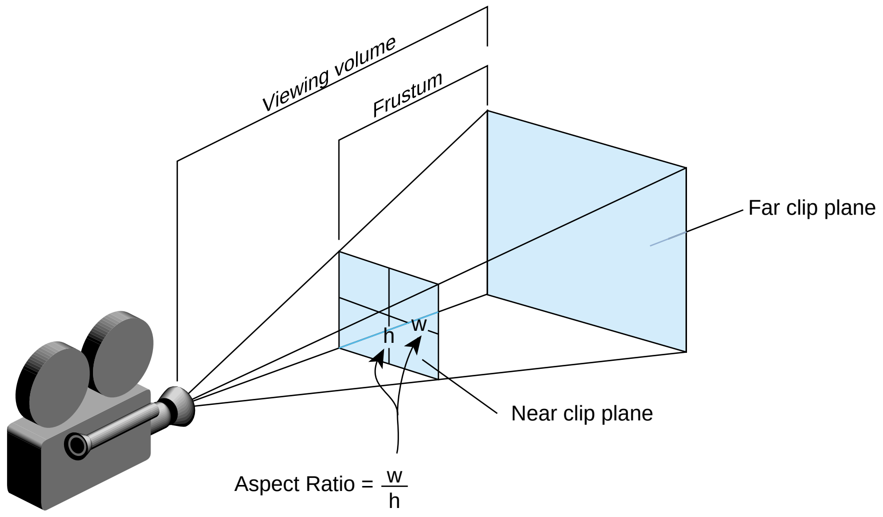

Figure 9-2 Aspect Ratio

The distances to the near and far clip planes are in the Z dimension in camera space. The viewing frustum can be transformed into world space using the position and orientation methods in **csCamera**.

Normally, the aspect ratio of the image matches that of the window. If the aspect ratio does not match that of the window, the image is distorted: it is either expanded or contracted along one or both axes, as shown in Figure 9-3.

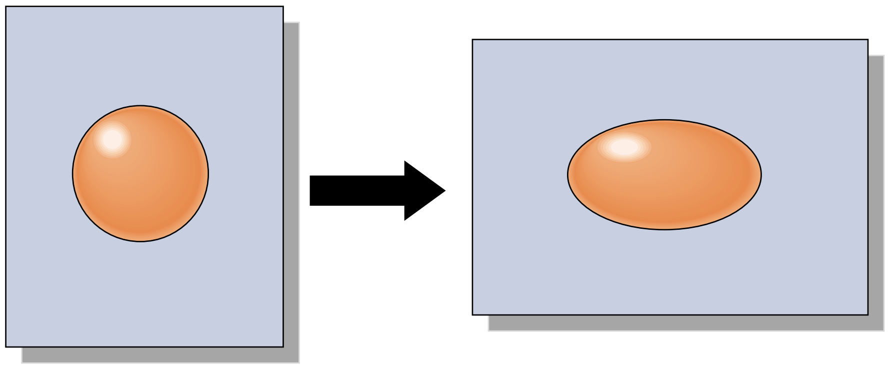

Figure 9-3 Changing the Window Without Changing the Image's Aspect

It is important, therefore, to change the aspect of the image if the window is revised. To do that, you use **csWindow**, as explained in Chapter 12, "User Interface Mechanisms." The aspect ratio can be set through methods of **csOrthoCamera** and **csPerspCamera**.

The following methods set the position and orientation of a camera.

```
void setPosition(const csVec3f& position);
void setPosition(csFloat v0, csFloat v1, csFloat v2);
void setOrientation(const csRotation& orientation);
void setOrientation(csFloat v0, csFloat v1, csFloat v2, csFloat v3);
```

There is a corresponding **get...()** field for each **set...()** field.

You use the **setPosition()** methods to locate the camera in the scene. Initially, it is pointing along the Z axis. To rotate the camera use the **setOrientation()** field. You set the near and far clip planes using the **setNear()** and **setFar()** methods.

csOrthoCamera

csOrthoCamera defines an orthographic projection. An orthographic projection uses a parallelepiped (box) frustum. Unlike the frustum in Figure 9-2, the size of the frustum does not change from one end to the other. For this reason, the distance from the camera to an object in the frustum does not affect the size of the object.

You use this type of camera when you do not want to view objects in perspective. For example, when you create architectural blueprints and CAD models, it is important to maintain the actual sizes of objects and angles between them when they are projected.

csOrthoCamera uses the following methods to define the viewing frustum:

```
void setWidth(csfloat width)
void setHeight(csfloat height)
void setCenter(const csVec2f* center)
void setCenter(csFloat v0, csFloat v1);
void setNearClip(csFloat nearClip);
void setFarClip(csFloat farClip);
```

There is a corresponding **get...()** field for each **set...()** field.

The width and height methods define the left, right, top, and bottom of the parallelepiped (box) frustum. The **setCenter()** method points the **csOrthoCamera** at the center of the scene you want to view.

csPerspCamera

A **csPerspCamera** creates a perspective view in which objects closer to the camera appear larger than the same-sized objects located further from the camera. This type of camera imitates normal vision. For example, train tracks and the distance between them appear smaller the more distant they are.

You can understand why more distant objects appear smaller by looking at Figure 9-4.

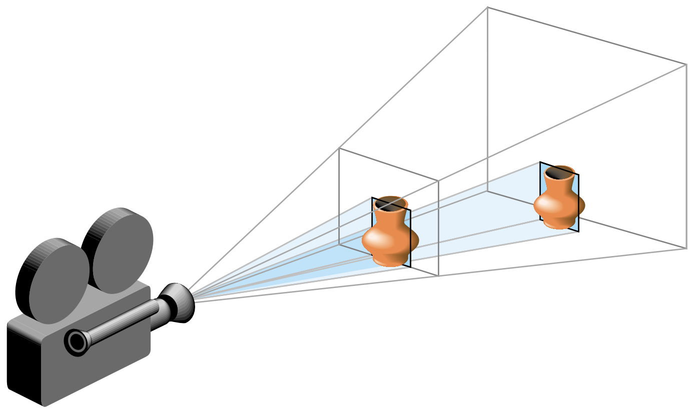

Figure 9-4 Perspective Explained

If you compare the height of an object in the near clipping plane to the height of the near clipping plane, you see that the ratio is larger than the ratio of the height of the same object in the far clipping plane to the height of the far clipping plane.

$$\frac{H_{object}}{H_{near}} > \frac{H_{object}}{H_{far}}$$

Consequently, when you compare the size of the object to its surroundings, it appears smaller in the distance.

Setting the Frustum

The frustum is the truncated pyramidal volume depicted in Figure 9-2. It is defined by four quantities, as shown in Figure 9-5:

- Distance to the near clip plane.
- Distance to the far clip plane.
- Horizontal field of view.
- Vertical field of view.

Setting the Clip Planes

You use the following fields to set the distance to the near and far clip planes:

```
void setNearClip(csFloat nearClip);
void setFarClip(csFloat farClip);
```

There is a corresponding **get...()** field for each **set...()** field.

Setting the Fields of View

You use the following fields to set the horizontal and vertical fields of view (FOV) of the frustum. The arguments are the angles, in degrees, of the fields of view.

```
void setHorizFOV(csFloat horizFOV);
void setVertFOV(csFloat vertFOV);
```

There is a corresponding **get...()** field for each **set...()** field.

Offsetting the Fields of View

By default, the fields of view are centered around the –Z axis. **csPerspCamera**, however, enables you to offset the fields of view both horizontally and vertically, as shown in Figure 9-5.

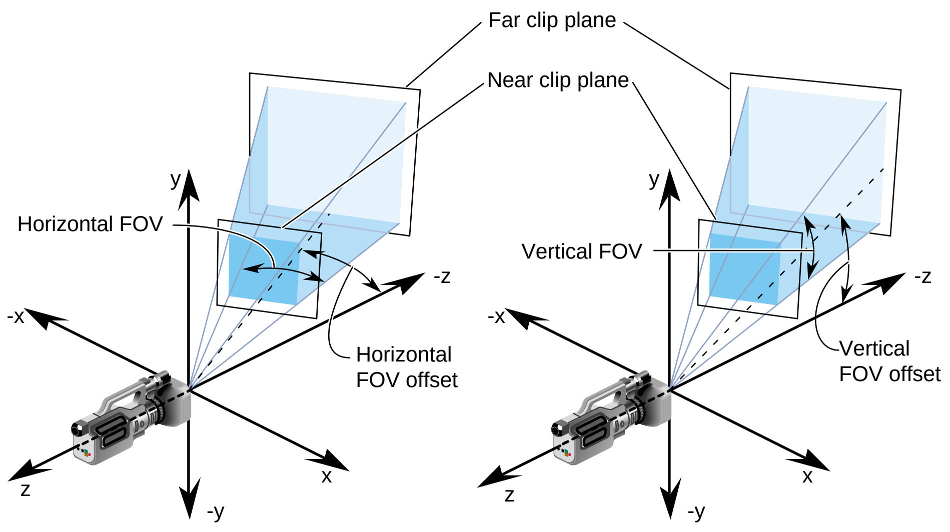

Figure 9-5 Horizontal and Vertical Fields of View Offsets

The offset angles are measured starting at the -z axis, following the right-hand rule. For example, the horizontal FOV offset angle shown in Figure 9-5 is positive.

To specify the offsets, use one or both of the following **csPerspCamera** methods:

```
void setHorizFOVOff(csFloat horizFOV);
void setVertFOVOff(rfcsFloat vertFOV);
```

There is a corresponding **get...()** field for each **set...()** field.

csFrustumCamera

csFrustumCamera allows you to work directly with a frustum without worrying about the specifics of a camera. You define the frustum using the **csFrustumCamera**'s only method:

```
setFrustum(csFrustum * frustum)
```

For a definition of **csFrustum**, see *csGeoMath.h*.

Chapter 10

Scene Graph Engines

There are classes that work with scene graphs, but are not nodes; they cannot be part of the scene graph, but they serve the vital function of enabling animation. **csEngine** is such a class.

This chapter describes **csEngine** and the multiple subclasses derived from it.

These are the sections in this chapter:

Engines

A **csEngine** performs a specified function on input data and outputs a result. These results can be used, for example, to encapsulate scene graph behavior. If part of a scene graph renders a car, you might, attach an engine to the transform node of each wheel to animate its rotation. You might also tie the motion of the wheels to the motion of other shapes in the scene so they appear to pass by when the car moves, or you might cycle the car through a set of colors repeatedly.

csEngine is a derivative of **csNode**, but usually **csEngines** are not included as nodes in a scene graph.

Input and Output Fields

csEngine has input and output fields. **csEngine** updates its output fields according to the function carried out by the **csEngine.** For example, a simple engine might take two inputs and output the average of the two.

csEngine updates its output fields only under the following conditions:

- Output fields are read.
- Input fields have changed since the output fields were last read.

For example, the input fields might change twice, but if the output fields are read only after the second change in the input fields, the output fields are updated only once.

Connecting Engines to Other Nodes

You connect an engine to other nodes or engines to create an animation, to cause a chain reaction of motions, or to cycle through a set of attributes, such as color cycling. You use **csContainer::connect()** to make the connections. For example, to make a color interpolator engine change a specific field in a material, connect the engine's output field value to the specific field, as follows:

```
csColorInterpolator* myEngine = new csColorInterpolator;
csMaterial* rose = new csMaterial;

myEngine->connect(csColorInterpolator::VALUE, rose,
    csMaterialDIFFUSE_COLOR);
```

In this example, *myEngine* is connected to the color field of the *rose* object.

Connecting Engines to Other Engines

The output of one engine may also be connected to the input of another engine. For example, the output value of a **csSpline** might be connected to the input field of a **csMorphEng**, as follows:

```
csMorphEng3f *morphEngine = new csMorphEng3f;
csSpline *splineEngine = new csSpline;

splineEngine->connect(csSpline::WEIGHT, morphEngine,
    csMorphEng3f::WEIGHT);
```

Engine Types

Many engines interpolate between two values at specified increments. For example, a rotation interpolator might take the beginning and ending rotation coordinates and the incremental changes between the two. Its output might be a series of transformations that move an object through a series of coordinates that rotate it from the beginning to the ending coordinates.

Another example for using an engine is color cycling. You might take an interpolator engine that takes a series of colors as inputs and outputs a color that gradually changes to from one input color to the next.

The following **csEngines** interpolate data:

- **csSpline—**interpolates an arbitrary, non-uniform spline and outputs a weighted array that defines a weight for each key in the spline.
- **csSelectorEng—**selects one coordinate from an array input as its single output. Derived classes include **csSelectorEng3f** and **csSelectorEng4f**.
- **csInterpolator—**interpolates between keyframe values selected from the key array by a floating point fraction, ranging from 0 to 1.
- **csColorInterpolator**—linearly interpolates among a set of colors.
- **csCoordinateInterpolator—**linearly interpolates among sets of coordinates, possibly generating more than one output to help produce, for example, a geometric morph.
- **csNormalInterpolator—**linearly interpolates among a set of normal vectors, possibly generating more than one output to help produce, for example, a geometric morph.
- **csOrientationInterpolator**—linearly interpolates among a set of rotations.
- **csPositionInterpolator—**linearly interpolates among a set of positions in space.
- **csScalarInterpolator**—linearly interpolates between a set of floats.

csInterpolator is the base class for the following engines:

- **csColorInterpolator**
- **csCoordinateInterpolator**
- **csNormalInterpolator**
- **csOrientationInterpolator**
- **csPositionInterpolator**
- **csScalarInterpolator**

The following **csEngines** can be used to change the features of a shape:

- **csMorphVec**—produces a weighted sum of attribute sets. Derived classes include **csMorphVec3f** and **csMorphVec4f**.
- **csTransformEng**—transforms attribute sets consisting of points or vectors (homogeneous coordinate implicitly 1 or 0 respectively). Derived classes include **csTransformEng3f** is derived from **csTransformEng.**

The following sections describe each of these nodes.

Engines that Interpolate Values

csInterpolator is an abstract base class that interpolates between keys, as shown in Figure 10-1. The keys might be location, normal, or rotation values.

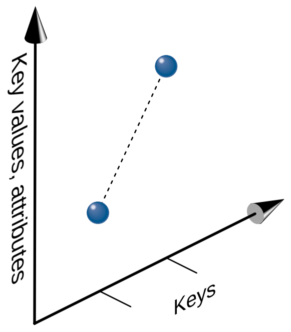

Figure 10-1 Keys and Key Values

Interpolator nodes are designed for linear, keyframed animation, that is, an interpolator node defines a piecewise linear function, **f(k)**, on the interval [-∞, ∞]. The piecewise linear function is defined by *n* keys and *n* corresponding key values, **f(k)**. The keys must be monotonic and non-decreasing. An interpolator node evaluates **f(k)** given any value of *k*.

Interpolator Engine Terminology

Interpolator engines take one or more input values, perform a function on them, and output a result. The relationship between the input and output values can be represented by a graph where the x axis represents the input values and the y axis represents the output values. Graphing the points (x_n, y_n) shows the shape of the engine's function.

Interpolator engine methods use the following terminology:

- keys—input values represented on the x axis.
- key values—output values represented on the y axis.
- **setFraction()**—a method in interpolator engines that specifies a point on the y axis.

Figure 10-2 illustrates these terms.

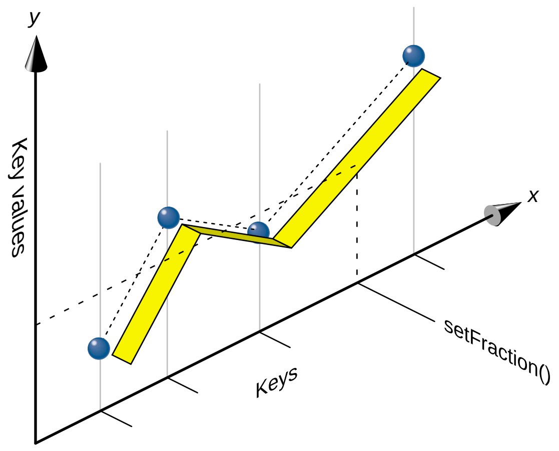

Figure 10-2 Engine Terminology

csInterpolator Fields

The fields in **csInterpolator** include:

```
void                setFraction(float fraction);
float               getFraction();

csMFFloat*          key();
```

For an explanation of **key()** and **setFraction()**, see, “Interpolator Engine Terminology” on page 111.

csSpline

A **csSpline** takes an array of keys as its input and outputs a set of weights. A *key* is one or more pieces of data; a *weight* is a fractional scaling factor. How the output is weighted depends on the order of the spline. Figure 10-3 shows two different splines that use the same key values, but are of different orders: piecewise linear and cubic.

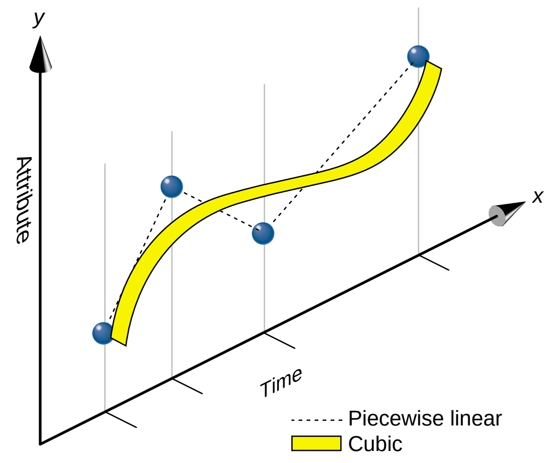

Figure 10-3 Spline

Note: Release 1.0 and 1.1 of Cosmo 3D only support the piecewise linear order.

Typically, the output of a **csSpline** is used as the input of a **csMorphEng**.

Keys and Key Values

In Figure 10-3, the X axis represents *keys*, such as time, and the Y axis represents any attribute or set of attributes, such as location, color, or transparency. Y axis values are called *key values*. Notice that the interval between the key values is nonuniform. After setting the key values, **csSpline** fills in the values between the key values.

The difference between the piecewise linear and cubic splines is created by the different weight factors.

Values outside of the maximum and minimum keys are clamped to the maximum and minimum keys and key values.

Key values, such as colors or coordinates, are not kept in a **csSpline** object. Consequently, a single **csSpline** object can define the animation spline for many keys. Key values are typically kept in a **csMorphEng** engine, which calculates the weighted sum of key values to produce the final result.

csSpline Fields

The following fields set the spline values:

```
csMFFloat*  key() const;
csMFFloat*  weight() const;

void        setFraction(csFloat fraction);
csFloat     getFraction();

void        setBasis(const csMatrix4f& basis);
void        getBasis(csMatrix4f& basis);
```

For an explanation of **key()** and **setFraction()**, see, “Interpolator Engine Terminology” on page 111.

weight() specifies the multiplication factor that changes the linear graph into a spline curve.

setBasis() specifies the matrix by which you multiply the four key values (closest to the fraction) to get the weight values.

Note: setBasis() is ignored in Cosmo 3D release 1.0 and 1.1. All splines are piecewise linear.

csColorInterpolator

This node interpolates among a set of key values in a **csMFColor** to produce a **csSFColor** (RGB) color.

csColorInterpolator contains the following fields:

```
csMFVec3f*   keyValue() const;
csSFVec3f    value
```

keyValue() is the set of RGB color values over which you want to interpolate. The number of colors in the **keyValue()** field must be equal to the number of keyframes in the key field.

For a further explanation of **keyValue()**, see “Interpolator Engine Terminology” on page 111.

csCoordinateInterpolator

This node linearly interpolates among a set of **MFVec3f** values. This would be appropriate for interpolating coordinate positions for a geometric morph.

csCoordinateInterpolator contains the following fields:

```
csMFVec3f*   keyValue() const;
csMFVec3f    value
```

keyValue() contains sets of positions used for the coordinate interpolation. The number of elements stored in the value field is equal to the number of elements in the **keyValue()** field divided by the number of elements in the key field.

For a further explanation of **keyValue()**, see “Interpolator Engine Terminology” on page 111.

csNormalInterpolator

csVec3f values can represent unit vectors normal to the surface of a unit sphere. **csNormalInterpolator** interpolates between normals.

The output values for a linear interpolation from a point P on the unit sphere to a point Q also on a unit sphere should lie along the shortest arc (on the unit sphere) connecting points P and Q.

When P and Q are pointed in opposite directions, they are on opposite sides of the unit sphere, and therefore all arcs connecting them on the unit sphere are the same length, so an infinite number of arcs describe the shortest path between the two points. The interpolation can be along any one of these arcs.

csNormalInterpolator contains the following fields:

```
csMFVec3f*   keyValue() const;
csMFVec3f    value
```

keyValue() contains the vectors used for the normal interpolation. The number of elements stored in the value field is equal to the number of elements in the **keyValue()** field divided by the number of elements in the key field.

For a further explanation of **keyValue()**, see, “Interpolator Engine Terminology” on page 111.

csOrientationInterpolator

A **csOrientationInterpolator** interpolates between two rotations by computing the shortest path on the unit sphere between the two rotations. The interpolation will be linear in arc length along this path.

If two vectors are pointed in opposite directions, they are on opposite sides of the unit sphere and therefore all arcs connecting them on the unit sphere are the same length, so an infinite number of arcs describe the shortest path between the two points. The interpolation can be along any one of these arcs.

csOrientationInterpolator contains the following fields:

```
csMFRotation*    keyValue() const;
csSFRotation     value
```

keyValue() contains the keys used for the coordinate interpolation. The number of elements stored in the value field is equal to the number of elements in the **keyValue()** field divided by the number of elements in the key field.

For more information about **keyValue()**, see, “Interpolator Engine Terminology” on page 111.

csPositionInterpolator

csPositionInterpolator linearly interpolates between sets of values in a **SFVec3f**. This is appropriate for interpolating a translation. The vectors are interpreted as absolute positions in local space. The **keyValue** field must contain exactly as many values as in the key field.

csPositionInterpolator Fields

csPositionInterpolator contains the following field:

```
csMFVec3f*   keyValue() const;
csSFVec3f    value
```

keyValue() contains the float values used for the scalar interpolation. The number of elements stored in the value field is equal to the number of elements in the **keyValue()** field divided by the number of elements in the key field.

For more information about **keyValue()**, see, "Interpolator Engine Terminology" on page 111.

csScalarInterpolator

csScalarInterpolator linearly interpolates between a set of **csSFFloat** values. This interpolator is appropriate for any parameter that is defined with a single floating point value, for example, width, radius, and intensity.

csScalarInterpolator contains the following fields:

```
csMFFloat keyValue[];
csSFFloat value;
```

For an explanation of **keyValue()**, see, "Interpolator Engine Terminology" on page 111.

csSelectorEng3F and csSelectorEng4F

csSelectorEng3F and csSelectorEng4F sets the output value to the nth item in the input array where n is the value of selector.

```
csMFVec3F input[];
csSFVec3F value;

csMFVec4F input[];
csSFVec4F value;
```

These classes are derived from the abstract, base class csSelectorEng, which provides the following methods for setting the selector:

```
void setSelector (int selector);
```

Engines That Change Shapes

The following engines generally use as input the output of one of the interpolator engines for the purpose of changing some feature of a **csGeoSet**.

csMorphEng

csMorphEng is an abstract class derived from **csEngine** that morphs a set of attributes from one setting to another, such as you might expect when one value incrementally changes to another. The output of this engine is a weighted sum of attributes, such as a set of coordinates. Any number of variably-sized attribute sets, however, can be packed into the single input field.

The input and output data are held in different vector arrays, such as a **csMFVec3f**, according to the dimensions of the attribute data.

csMorphEng is an abstract class. Subclasses in Cosmo 3D include **csMorphEng3f** and **csMorphEng4f**. The only difference between the classes is the number of attributes transformed.

csMorphEng Fields

csMorphEng contains the following fields:

```
csMFInt*            count() const;
csMFInt*            index() const;
csMFFloat*          weight() const;
```

count() is an array of integer values.

Each value represents the number of input values which will be morphed based on the corresponding weight value. If the value of **count()** is positive, the next coordinate is chosen by dereferencing and incrementing the input array pointer. If the value of **count()** is negative, the index of the next coordinate is chosen by dereferencing and incrementing the index array pointer, and that index is used to choose the coordinate from the input array.

weight() is an array of float values.

For each weight, **abs(count())** coordinates are multiplied by weight and stored in the output array.

index() is an optional carry of indices used to index into the input array when generating output values.

See the description of **count()** above for more information.

csMorphEng3f and csMorphEng4f

csMorphEng3f and **csMorphEng4f** are derived from **csMorphEng**. Their input and output values are **csMFVec3f** and **csMFVec4f,** respectively.

Example 10-1, taken from the test program, *worm.cxx*, shows how to build a **csMorphEng3f** engine.

Example 10-1 Building a Morph Engine: the Worm

```
// Build morph engine
    MorphCoords = new csMorphEng3f;
    // “Neutral” is non-indexed and always has weight of 1
    MorphCoords->input()->setRange(0, NumRings*RingVerts, coords);
    MorphCoords->count()->set(0, NumRings*RingVerts);
    MorphCoords->weight()->set(0, 1.0f);

    // build the coordinate vectors for the rings on the worm
    for (k=0,i=0; i<NumRings; i++)
    {
        for (j=0; j<RingVerts; j++,k++)
        {
            csVec3f     v(coords[k]);

            v.scale(.5f, v);

            // Set displacement vectors for “target” i
            MorphCoords->input()->set(k+NumRings*RingVerts, v);
            MorphCoords->index()->set(k, k);
        }
        // targets are indexed
        MorphCoords->counts()->set(i+1, -RingVerts);
    }
    MorphCoords->connectOutput(csMorphVec3f::OUTPUT,
                               SkelCoords, csTransformEng3F::INPUT);
```

This engine moves the rings in the worm.

csTransformEng

csTransformEng is a **csEngine** that translates attribute sets consisting of points or vectors. The **csTransformEng** output is a single array which may be used as a **csGeoSet** attribute list, for example, **csCoordSet3f**, **csNormalSet3f.**

Any number of variably-sized attribute sets can be packed into the input field.

csTransformEng is an abstract class. Subclasses in Cosmo 3D include **csTransformEng3f**. **csTransformEng3f** input and output values are **csMFVec3f**s.

csTransformEng Fields

csTransformEng has the following fields:

```
void                setTransformType(TransformTypeEnum transformType);
TransformTypeEnum   getTransformType();

csMFInt*            count() const;
csMFInt*            index() const;
csMFMatrix4f*       matrix() const;
```

transformType() specifies the input data type. If the **transformType()** value is

- POINT—the input is interpreted as points with a homogeneous value of 1.0.
- VECTOR—the input is interpreted as vectors with a homogeneous value of 0.0 which are transformed by the inverse transpose of the matrices in **matrix()**.

matrix() is an array of matrix values. For each weight, **abs(count())** coordinates are multiplied by the weight and stored in the output array.

count() is an array of integer values. Each value represents the number of input values which will be morphed based on the corresponding matrix value. If the value of **count()** is positive, the next coordinate is chosen by dereferencing and incrementing the input array pointer. If the value of **count()** is negative, the index of the next coordinate is chosen by dereferencing and incrementing the index array pointer, and that index is used to choose the coordinate from the input array.

Example 10-2 creates a sample **csTransformEng3f**.

Example 10-2 Creating a csTransformEng3f

```
// Build transform engine
SkelCoords = new csTransformEng3f;
SkelCoords->setTransformType(csTransformEng3f::POINT);
SkelCoords->input()->setRange(0, NumRings*RingVerts, coords);
```

This transformation engine positions and orients the rings on the worm in *worm.cxx*.

Chapter 11

Sensors

Sensors are used to detect one of the following:

- Time passing.
- Pointer device events.

Sensors are often implemented to perform one of the following tasks:

- Trigger an engine at a specified interval (**csTimeSensor**).

  This task is most commonly used to animate geometries.
- Generate rotations based on pointer device events (**csSphereSensor**).
- Generate translations based on pointer device events (**csPlaneSensor**).
- Determine whether the pointer device cursor is over a geometry (**csTouchSensor**).

This chapter explains how to implement sensors in the following sections:

- "csTimeSensor" on page 122.
- "csSphereSensor" on page 126.
- "csPlaneSensor" on page 130.
- "csTouchSensor" on page 134.

csTimeSensor

csTimeSensor generates timer events either once or repeatedly for a specified interval. A **csTimeSensor** is typically used to drive animations or periodic events.

A **csTimeSensor** has five inputs:

- enabled
- loop
- startTime
- stopTime
- cycleInterval

A **csTimeSensor**'s has four output events:

- cycleTime
- fractionChanged
- isActive
- time

Enabling csTimeSensor

To generate time sensor events, you must first enable **csTimeSensor** by setting the argument of the following method to TRUE:

```
csTimeSensor::setEnabled();
```

If the argument evaluates FALSE, time sensor events are not generated.

To determine whether or not a **csTimeSensor** is enabled, use the following method:

```
csTimeSensor::getEnabled();
```

Updating csTimeSensor

csTimeSensors are not automatically evaluated. The following method triggers the evaluation of all instantiated **csTimeSensors**:

```
csTimeSensor::updateSensors();
```

This method should be called at regular intervals, usually once per frame, for **csTimeSensors** to function as expected.

Updating with csWindow

By default, if there are instantiated **csTimeSensors**, **csWindow** calls **csTimeSensor::updateSensors()** at regular intervals; the default interval is 16 milliseconds. The frequency of updates can be modified using the following method:

```
csWindow::enableTimer(float ms);
```

where the argument, *ms*, specifies the frequency of updates in milliseconds. This method enables a timer event, which is fired every *ms* milliseconds. When the timer event is handled **csWindow** calls **csTimeSensor::updateSensors()**.

The timer event can be disabled by calling:

```
csWindow::disableTimer();
```

Note: The Optimizer class, **opViewer**, also calls **csTimeSensor::updateSensors()** by default if there are instantiated time sensors. **opViewer** also contains the methods **enableTimer()** and **disableTimer()** for enabling and disabling timer events.

Setting the Start and Stop Times

csTimeSensor::startTime() and **csTimeSensor::stopTime()** are **csTime** values representing the times at which the **csTimeSensor** should start and stop generating time sensor events.

If *stopTime* is less than *startTime*, the **csTimeSensor** generates events indefinitely, or terminates at the end of one cycle if **csTimeSensor:loop()** is FALSE. For more information about **loop()**, see “Continuing Timer Events” on page 125.

You can, for example, trigger events based on the time of day using **csTime::getTimeOfDay()**, which returns the current time of day. Your application can then determine whether or not that time falls between *startTime* and *stopTime*.

The following methods are used to get and set the values of *startTime* and *stopTime*:

```
csTimeSensor::getStartTime();
csTimeSensor::setStartTime();
csTimeSensor::getStopTime();
csTimeSensor::setStopTime();
```

isActive

A **csTimeSensor** generates an isActive event when the **csTimeSensor** is first activated or deactivated. A **csTimeSensor** is activated if:

- Time sensors are updated.
- The current time is between *startTime* and *stopTime*.

A **csTimeSensor** is deactivated when the current time is past *stopTime* or when the argument of **loop()** is FALSE and the time sensor has completed one cycle.

You can retrieve the current value of isActive by calling:

```
csTimeSensor::getIsActive();
```

Setting Cycle Duration

cycleInterval is a float representing the duration of a single cycle in seconds.

For example, if **csTimeSensor** is driving an animation, a cycle generally represents the number of times you want an animation to run. To run an animation once, set *cycleInterval* to the length of the animation. To run an animation twice, set *cycleInterval* to twice the length of the animation.

The following two methods are used to get and set the value of *cycleInterval*:

```
csTimeSensor::getCycleInterval();
csTimeSensor::setCycleInterval();
```

Continuing Timer Events

csTimeSensor::loop() takes a boolean value. If it is TRUE, the **csTimeSensor** continues to generate events after the completion of one cycle. If the boolean value is FALSE, **csTimeSensor** only generates events for a single cycle.

The following two methods are used to get and set the value of **loop()**:

```
csTimeSensor::getLoop();
csTimeSensor::setLoop();
```

Cycle Time Event

A **csTimeSensor** generates a *cycleTime* event whenever the **csTimeSensor** begins a new cycle. *cycleTime* is an absolute **csTime** value. If the **csTimeSensor** runs for only a single loop, the *cycleTime* event is only generated at the beginning of the loop.

The current value of *cycleTime* can be retrieved by calling:

```
csTimeSensor::getCycleTime();
```

Fraction Changed Event

A **csTimeSensor** generates a *fractionChanged* event on every update while the **csTimeSensor** is active. *fractionChanged* is a float value in the range [0.0, 1.0] representing the percentage of the current cycle completed. If the argument of **loop()** is TRUE, at the beginning of the first cycle the value of *fractionChanged* is 0.0. However, the beginning of all subsequent cycles generate a value of 1.0. The different value is a result of the beginning of one cycle coinciding with the end of the previous cycle.

The value of *fractionChanged* is returned using the following method:

```
csTimeSensor::getFractionChanged();
```

csSphereSensor

csSphereSensor maps the drag motion of a pointer device into a spherical rotation about a virtual sphere. The rotation gives the impression that the geometry is attached to the surface of a spinning sphere.

Virtual Sphere

The center of the virtual sphere is located at the center of either the local or world coordinate system, based on whether the argument of **csSphereSensor::coordFrame()** is LOCAL or WORLD, respectively.

The virtual sphere's radius is determined by the distance between the center and the initial point of intersection.

Offsetting the Rotation

You can offset the rotation of the virtual sphere using the following method:

```
csSphereSensor::setOffset(csRotation offset);
```

This method adds *offset* to the rotation value derived from the motion of the pointer device.

setAutoOffet() sets the value of *offset* in **setOffset()** to the value of the rotation when the **csSphereSensor** is deactivated.

To enable autoOffset, call the following method with an argument that evaluates TRUE:

```
csSphereSensor::setAutoOffet(csBool autoOffset);
```

In the case where a **csSphereSensor** is used to rotate its associated geometry, setting autoOffset to TRUE keeps the geometry from snapping back to its original orientation when the **csSphereSensor** is activated a second time.

csSphereSensor Events

A **csSphereSensor** begins generating events when both of the following conditions are met:

- The pointer device cursor is depressed over geometry associated with a **csSphereSensor**.
- **csSphereSensors** are updated to determine whether the pointer device cursor is depressed while over a geometry associated with a **csSphereSensor**.

While the **csSphereSensor** is active, it generates rotation and trackPoint events. Rotation events represent the orientation of a geometry on the virtual sphere. trackPoint represents the position of the pointer device cursor on the virtual sphere, as shown in Figure 11-1.

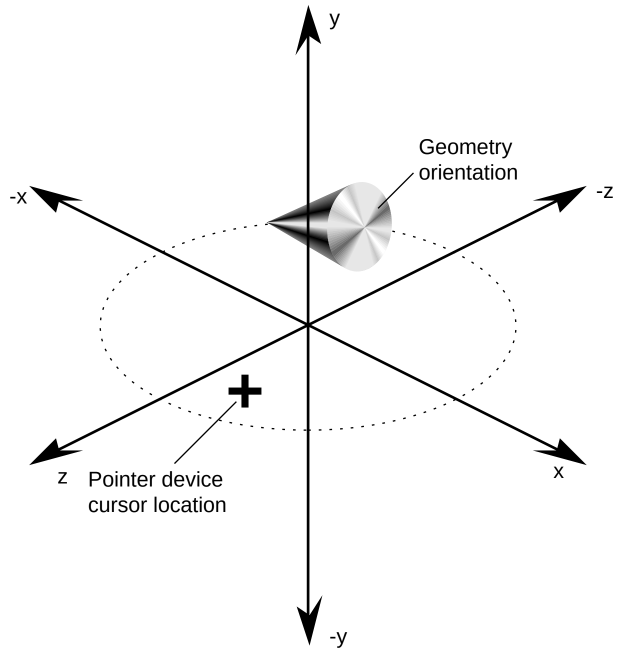

Figure 11-1 Rotation and trackPoint Representations

The **csSphereSensor** stops generating events when the pointer device is released.

Updating csSphereSensor

csSphereSensor is not updated automatically. You customarily update it explicitly when pointer device events are detected, using one of the following methods:

```
csSphereSensor::updateSensors();
csWindow::updateSensors();
```

The difference between the methods is that the arguments to the **csSphereSensor** version includes pointer device coordinates.

Note: **csSphereSensor** inherits **updateSensors()** from its abstract, base class, **csPickSensor**.

Setting Up csSphereSensor

The typical use of a **csSphereSensor** is to connect it with a transformation node. The effect of the sensor when combined with a transformation node is to rotate associated geometries spherically in local or world space according to the dragging motion of a pointer device: dragging the pointer device to the right spins the front of the geometry to the right; dragging the pointer device up spins the front of the geometry up.

Scope of csSphereSensor

A **csSphereSensor** is associated with all geometry in the scene graph “below” the parent node of the **csSphereSensor**, as shown in Figure 11-2.

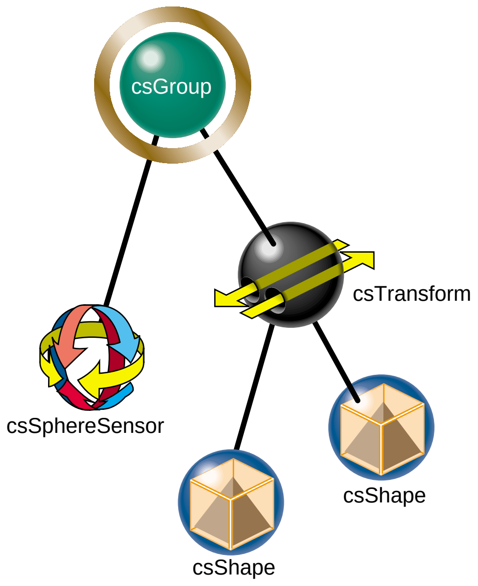

Figure 11-2 Placing csSphereSensor in a Scene Graph

A **csSphereSensor** is activated when the pointer device is depressed over its associated geometry. However, its output events, rotation and trackPoint, can be connected to any field in a scene graph. The effects of **csSphereSensor**, therefore, are not limited to its associated geometry.

Rotating Geometry Using csSphereSensor

To enable **csSphereSensor** to rotate geometry spherically, you must complete the following two tasks:

1. Connect a **csSphereSensor** node to a **csTransform** node, as follows:

   ```
   csSphereSensor *sphereSensor = new csSphereSensor;
   csTransform *transform = new csTransform;

   sphereSensor->connect(csSphereSensor::ROTATION, transform,
       csTransform::ROTATION);
   ```

   Rotation events are generated by **csSphereSensor**.

2. Add the **csSphereSensor** node as a child to the parent node of the transformation node associated with the geometry to rotate, as shown in Figure 11-2.

csPlaneSensor

A **csPlaneSensor** maps the drag motion of a pointer device's events into a translation in the XY plane of the local or world coordinate space, depending on the value specified by **csPlaneSensor::coordFrame()**.

Setting Up csPlaneSensor

A typical use of **csPlaneSensor** is to connect its output event, translation, to a transformation node. The effect of the sensor when combined with a transformation node is to translate associated geometry in local or world space along the XY plane according to the dragging motion of a pointer device: dragging the pointer device to the right moves the geometry to the right; dragging the pointer device up moves the geometry up.

To enable **csPlaneSensor** to translate geometry, you must complete the following two tasks:

1. Connect a **csPlaneSensor** node to a **csTransform** node, as follows:

   ```
   csPlaneSensor *planeSensor = new csPlaneSensor;
   csTransform *transform = new csTransform;

   planeSensor->connect(csPlaneSensor::TRANSLATION, transform,
       csTransform::TRANSLATION);
   ```

   Translation events are generated by **csPlaneSensor**.

2. Add the **csPlaneSensor** node as a child to the parent node of the transformation node associated with the geometry to translate, as shown in Figure 11-2.

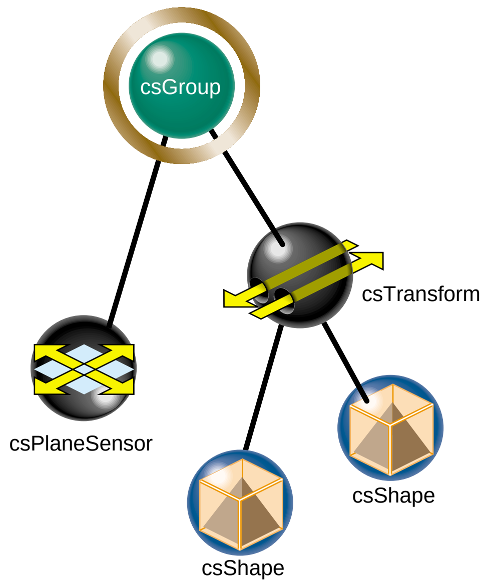

Figure 11-3 Placing csPlaneSensor in a Scene Graph

Scope of csPlaneSensor

A **csPlaneSensor** translates all geometries in the scene graph “below” the parent node of the **csPlaneSensor**.

A **csPlaneSensor** is activated when the pointer device is depressed over its associated geometry. However, its output events, translation and trackPoint, can be connected to any field in a scene graph. The effects of **csPlaneSensor**, therefore, are not limited to its associated geometry.

csPlaneSensor Events

A **csPlaneSensor** begins generating events when both of the following conditions are met:

- The pointer device cursor is depressed over geometry associated with a **csPlaneSensor**.
- **csPlaneSensors** are updated to determine whether the pointer device cursor is depressed while over a geometry associated with a **csPlaneSensor**.

The **csPlaneSensor** stops generating events when the pointer device is released.

Updating csPlaneSensor

csPlaneSensor is not updated automatically. You customarily update it explicitly when pointer device events are detected, using one of the following methods:

```
csPlaneSensor::updateSensors();
csWindow::updateSensors();
```

The difference between the methods is that the **csPlaneSensor** version includes pointer device coordinates.

Note: **csPlaneSensor** inherits **updateSensors()** from the abstract, base class, **csPickSensor**.

Limiting Translations

While the **csPlaneSensor** is active, it generates translation and trackPoint events. Translation events represent two-dimensional translations in the XY plane based on the motion of the pointer device. The translation value is clamped to the range defined in the following methods:

```
csPlaneSensor::minPosition();
csPlaneSensor::maxPosition();
```

If the X or Y component of *maxPosition* is equal to the corresponding component in *minPosition*, that component is constrained to the specified value.

Unclamped Translations

You can create unclamped translations in two ways:

- Setting *maxPosition* component values less than *minPosition* values.
- Using trackPoint events.

If the X or Y component values of **csPlaneSensor::maxPosition()** are less than the corresponding component values in **csPlaneSensor::minPosition()**, the translations in that component's direction are unclamped.

TrackPoint events represent unclamped drag positions of the geometry in the XY plane of local or world space, depending on the value set in **csPlaneSensor::setCoordFrame()**. For more information about **setCoordFrame()**, see "Local or World Translations" on page 133

To return trackPoint events, use the following method:

```
csPlaneSensor::getTrackPoint();
```

Local or World Translations

The translation of geometry associated with a **csPlaneSensor** occurs either in local or world space according to the value set in the following method:

```
csPlaneSensor::setCoordFrame();
```

The argument of the method can be either LOCAL or WORLD.

csPlaneSensor Offsets

You can offset the translation of the geometry using the following method:

```
csPlaneSensor::setOffset(csVec3f offset);
```

This method adds *offset* to the translation value derived from the motion of the pointer device.

setAutoOffet() sets the value of *offset* in **setOffset()** to the value of the translation when the **csSphereSensor** is deactivated.

To enable autoOffset, call the following method with an argument that evaluates TRUE:

```
csPlaneSensor::setAutoOffset(csBool autoOffset);
```

In the case where a **csPlaneSensor** is used to translate its associated geometry, setting autoOffset to TRUE keeps the geometry from snapping back to its original position when the **csPlaneSensor** is activated for a second time.

csTouchSensor

csTouchSensor tracks the location of the pointer device cursor and generates up to five outputs when the cursor is over the geometry associated with **csTouchSensor**. If the pointer device cursor passes over multiple geometry, events are generated based on the geometry nearest the camera.

Associating csTouchSensor and Geometry

A geometry is associated with a **csTouchSensor** when the geometry and **csTouchSensor** nodes are descendants of the same node, as shown in Figure 11-4.

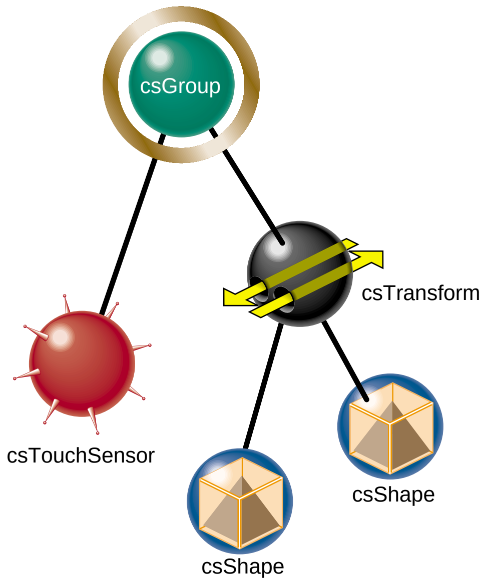

Figure 11-4 Placing csTouchSensor in a Scene Graph

Scope of csTouchSensor

A **csTouchSensor** monitors all geometries in the scene graph “below” the parent node of the **csTouchSensor**. Not only does it monitor the children of **csTouchSensor**’s parent node, it also monitors all the descendants of the children.

csTouchSensor Output

csTouchSensor generates up to five of the following events, each with their own preconditions:

- isOver
- hitPoint
- hitNormal
- hitTexCoord
- touchTime

isOver Event

isOver returns a boolean value indicating whether the pointing device is over a geometry associated with a **csTouchSensor**. **csTouchSensor** generates the isOver event when the isOver state has just changed.

The isOver state changes whenever the pointer device cursor first passes over geometry associated with a **csTouchSensor** or whenever the pointer device first passes off of the geometry it was previously over. The isOver event is not generated constantly while the pointer device is over a geometry.

When the pointer device cursor is over an associated geometry, the following method evaluates TRUE:

```
csTouchSensor::getIsOver();
```

Hit Events

csTouchSensor generates the following three hit events whenever the pointing device is over a geometry associated with a **csTouchSensor**:

- hitPoint—represents the coordinates where the pointer device cursor intersects the geometry.
- hitNormal—represents the surface normal vector where the pointer device cursor intersects the geometry.
- hitTexCoord—represents the surface texture coordinates where the pointer device cursor intersects the geometry.

touchTime Events

The touchTime event represents the current time. **csTouchSensor** generates touchTime events when any of the following conditions are true:

1. The pointer device cursor is over the geometry when the pointer device button is depressed.
2. The pointer device cursor is currently over the geometry and isOver is TRUE.
3. The pointer device button is released.

Chapter 12

User Interface Mechanisms

Cosmo 3D applications either appear within a **csWindow** object or a window object that you create using X window code. The window provides an interface for users to interact with the Cosmo 3D application.

Cosmo 3D also supports user interaction by enabling the selection of screen objects.

This chapter discusses how to implement user interaction using X window code, **csWindow**, and selection mechanisms.

These are the sections in this chapter:

Creating a csWindow

All of the **csWindow** fields have default values. You may find that they satisfy the needs of your application. *worm.cxx*, for example, uses all the default values.

If you want to change the initial position, size, and mode of the **csWindow** object, you use the following methods:

```
static void        initDisplayMode(unsigned long mode);
static void        initPosition(int x, int y);
static void        initSize(int width, int height);
```

To reposition or reshape the window after its initial display, use the following methods:

```
static void        positionWindow(int x, int y);
static void        reshapeWindow(int width, int height);
```

To specify the title of the window or its icon, use the following methods:

```
static void        setWindowTitle(const char *title);
static void        setIconTitle(const char *title);
```

Note: If the title is NULL, the window has no borders, except on the PC.

To show, hide, or iconify a window, use the following methods:

```
static void        iconifyWindow(void);
static void        showWindow(void);
static void        hideWindow(void);
```

To create a full screen window, set the position to (0, 0) and the size to the size of the screen. To get the screen dimensions, use **csWindow::get()**.

Manipulating the Window Stack

You can create more than one **csWindow** object at a time. You control the display of the **csWindow** objects by manipulating the window stack: the static **csWindow** methods manipulate the **csWindow** that is on the top of the stack.

The following methods in **csWindow** manipulate the window stack:

- **popWindow()** raises the current window to top of the window stack.
- **pushWindow()** lowers the current window to bottom of the window stack.

Manipulating the Window Buffers

Cosmo 3D uses two buffers:

- Front buffer—contains the graphic information currently being displayed.
- Back buffer—stores the next image to be displayed.

When the rendered image is ready to be displayed, you switch the source of the graphic information from one buffer to the other using either **swapBuffers()** or **swapThisWindowBuffers()**.

Handling User Input

Events, such as mouse motion, key presses, and window resize, are converted to **csEvent** objects and queued on their associated **csWindow**. When the application is ready to process these events the **getEvents()** method of each **csWindow** can be called to get the array of queued **csEvent**'s. After processing the events, the event array should be discarded by calling the window's **resetEvents()** method.

Using Callback Functions

Retrieving and handling user events using **getEvents()** requires constant polling by the application, which can waste processor time. To avoid this polling, a Cosmo 3D application typically sets event callbacks on **csWindow** objects and calls the static **csWindow::mainLoop()** method to handle all event processing. Using callbacks, the application is notified when events are waiting to be processed.

To set up event callbacks, an application can use either **csWindow::setFrameFunc()** or the newer (recommended) methods, **setEventFunc()**, **setRenderFunc()**, and **setFreeRun()**.

The user-defined, callback's return value **of setFrameFunc()** controls the manner in which events are processed. The return values include:

- BLOCK—the event processing loop blocks until another event occurs on some window.

  In BLOCK mode, the callback is called for every event, but not at all when no events are generated.
- CONTINUE—the callback is called every time the main loop goes idle.

  In CONTINUE mode, events are queued and delivered when things go idle. The callback continues to be called even when no events are generated.

Each window has its own "frameFunc" and that function is called when there are events on that window. This model gets confusing when multiple windows use the CONTNUE mode.

setEventFunc() defines a per-window event callback similar to **setFrameFunc()**, but the mode of event handling is controlled by **csWindow::setFreeRun()**. When there are no more events to process a global callback, **setRenderFunc()** is called indicating the windows should be re-rendered.

When **csWindow::setFreeRun()** is TRUE, callbacks are invoked only when the main loop goes idle, similar to the CONTINUE mode. Otherwise, callbacks occur with every event but nothing happens when the main loop is idle.

With the either of these event callback mechanisms, the event information, returned by **getEvents()**, is extracted from the **csEvent** objects. The application, however, does not need to call **resetEvents()**; it is called automatically.

Querying Devices

In addition to the above **csEvent**-centered model, you can use the methods **getMouseX()**, **getMouseY()**, **getMouseButtons()**, and **getDevice()** to query the current value of each user input device. This approach is discouraged.

Selecting Screen Objects

Cosmo 3D enables the selection of screen objects in the following ways:

- **csIsectAction—**selects the shape closest to the origin of a ray that is intersected by that ray.
- **csCamera::pick()—**selects the shape closest to the camera based on window coordinates.

These methods use a **csHit** object for storing the selected objects.

Using csIsectAction

csIsectAction, when applied to the root node selects all graphical objects intersected by a ray stored in a **csSeg**, as shown in Figure 12-1.

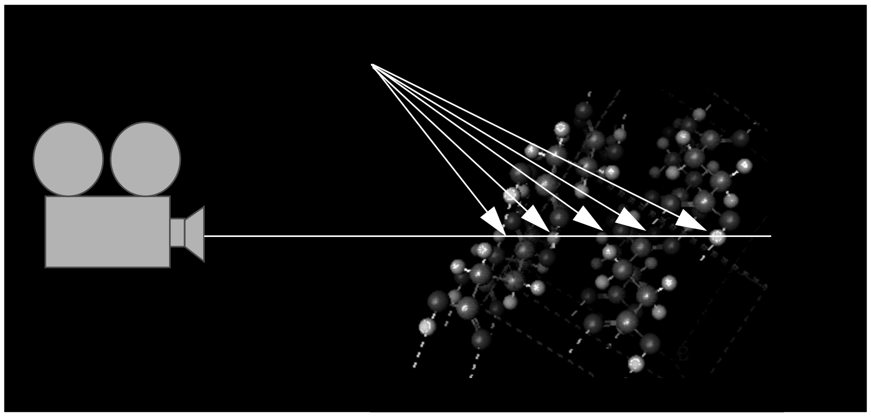

Figure 12-1 Ray Pick Action

The shape closest to the origin of the **csSeg** and intersected by the line segment is recorded in a **csHit** object. For information about **csHit**, see “Storing Selected Screen Objects” on page 142.

Using Pick()

csCamera::pick() uses window and viewport coordinates to select the screen object closest to the ray connecting the camera to the point. You might supply the window coordinates using the user input methods; see “Handling User Input” on page 139.

When you supply **pick()** with window coordinates, the method internally uses **csContext::getViewport()** to convert the coordinates to viewpoint coordinates. The method then calls **csIsectAction** to construct a ray from the camera to the coordinates. The screen object closest the camera on the ray is recorded in a **csHit** object. For information about **csHit**, see “Storing Selected Screen Objects” on page 142.

For information about **csContext::getViewport()**, see “Setting the Screen Display of the Scene” on page 97.

Storing Selected Screen Objects

csHit objects hold pointers to objects selected using a variety of mechanisms. The methods in **csHit** allow you to access the information held by **csHit**, including:

- The index number, *geomPartNumber*, of the triangle, quadrilateral, or polygon inside a **csGeoSet**.
- The **csGeometry** that was intersected.
- The **csShape** that was intersected.
- The normal in local space at the intersection point. (The space of the geometry that was intersected.)
- The intersection point in local space. (The space of the geometry that was intersected.)
- The model view matrix for the **csGeometry** intersected. The model view is the concatenation of the viewing matrix and all the matrices in all **csTransform** objects above the **csShape** node.
- The normal in world space at the intersection point. (The space of the camera used to calculate the hit.)
- The list of nodes leading from the root of the scene graph to the intersected **csGeometry**.
- The intersection point in world space. (The space of the camera used to calculate the hit.)
- Distance of the intersection from the **csSeg** origin, which is the same as the distance from the **csCamera** object.
- The line segment, **csSeg**, expressed in three dimensions, that was used in the intersection test.

Creating Your Own Window

Instead of using the window provided by Cosmo 3D, **csWindow**, you can create your own window using X11 or WIN32 window code. In this case your application controls the **csContext**, events, and window, as shown in Figure 12-2.

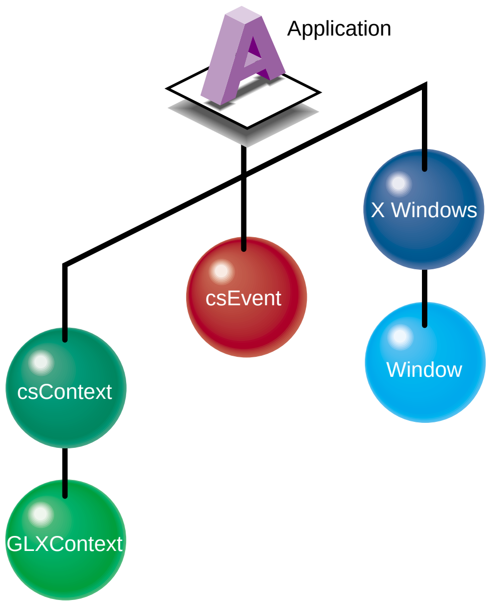

Figure 12-2 Creating Your Own Window

Sample Window Code

For sample code that shows you how to create your own window, see the demonstration programs, *cubex.cxx* and *cubew32.cxx*.

Chapter 13

Multiprocessing

If you would like to display two views of the same scene graph, you need to use the multiprocessing capabilities of Cosmo 3D. You might, for example, like to display a ground-view and an over-head view of the same scene at the same time. You can accomplish this by creating a thread for each view and connecting each to their own **csWindow** and **csContext**, as shown in Figure 13-1.

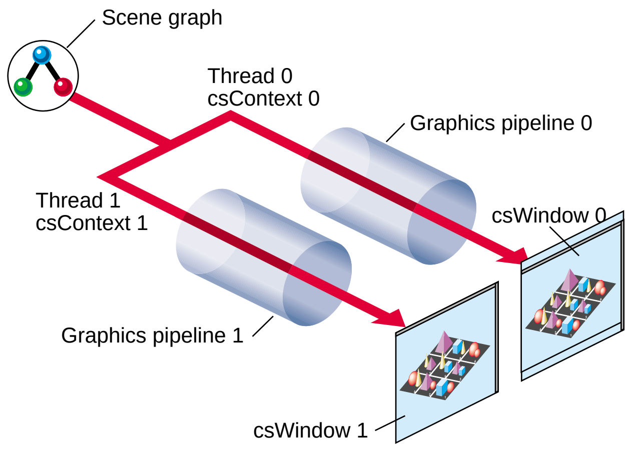

Figure 13-1 Multiprocessing

This chapter describes how to implement multiprocessing in the following sections:

- "Implementing Multiprocessing" on page 146.
- "Thread Blocking" on page 148.
- "Multithreaded Example" on page 150.

Implementing Multiprocessing

One rule that you must follow when implementing multiprocessing is the scene graph cannot be modified during rendering. Cosmo 3D does not provide a mechanism for concurrent scene graph modification and drawing.

The consequence of this rule is that you must implement semaphores to block the action of the application or draw threads.

Note: OpenGL Optimizer provides the semaphore class, **opBarrier**.

The following sections describe how to implement threads.

Creating Threads

For each view of the scene graph you want to display, you use a different thread. Each thread is bound to a **csWindow** and a **csContext**.

Threads must be created using **csThread**. Threads created in other ways produce unpredictable results.

A thread can only have one context attached to it at a time. Conversely, each thread that attempts to draw must have a separate context. Unpredictable results occur if you try to share a context with more than one thread.

A context attached to one thread cannot be used by another thread until the first thread releases it using **csContext::releaseCurrent()**.

To create a thread and bind it appropriately, use the following procedure:

1. Create a thread, as follows:

   ```
   csThread* drawThread = new csThread(drawThreadEntry, NULL);
   ```

2. Bind the thread to a **csWindow** and **csContext** using **csContext::makeCurrent()**, as follows:

   ```
   static void
   drawThreadEntry(void* arg)
   {

       csWindow    *theWindow;
       csContext   *theContext;

       theContext = theWindow->getThisWindowContext();

       theContext->makeCurrent(theWindow);
       // Context is now current; scenes will be drawn in theWindow.
   ...
   }
   ```

Starting Threads

Whether or not a thread runs immediately depends on the **csThread** constructor you use. The first form, **csThread()**, does not start executing. This behavior allows you to load thread-related parameters before using **csThread::start()** to actually start the thread.

The second form of the constructor, **csThread(Entry*** *entryPoint*, **void* arg=NULL)**, allows you to load thread-related parameters in the argument. For that reason, it begins execution immediately.

Thread Parameters

Thread-related parameters include the following **csThread** methods:

- **numProcessors()**—returns the number of usable processors on the system.
- **setStackSize()—**sets the stack size to be used for this thread when it starts.

Exiting Threads

A thread only terminates execution when it calls **csThread::exit()** on itself. Cosmo 3D does not support termination of threads by other threads.

Additional Thread Controls

The following **csThread** methods control the execution of threads:

- **sleep()—**suspends the calling thread for the number of microseconds specified in the argument.
- **wait()—**waits for this thread to terminate.

Thread Blocking

In general, a Cosmo 3D application has two kinds of threads:

- The application thread, which handles events and creates draw threads.
- Multiple draw threads, which render different views of the scene graph.

In general, when one kind of thread is active, the other must be blocked. So while the application thread handles events and modifies the scene graph, the draw threads must be blocked and, conversely, while the draw threads are active, the application thread must be blocked from modifying the scene graph, as shown in Figure 13-2.

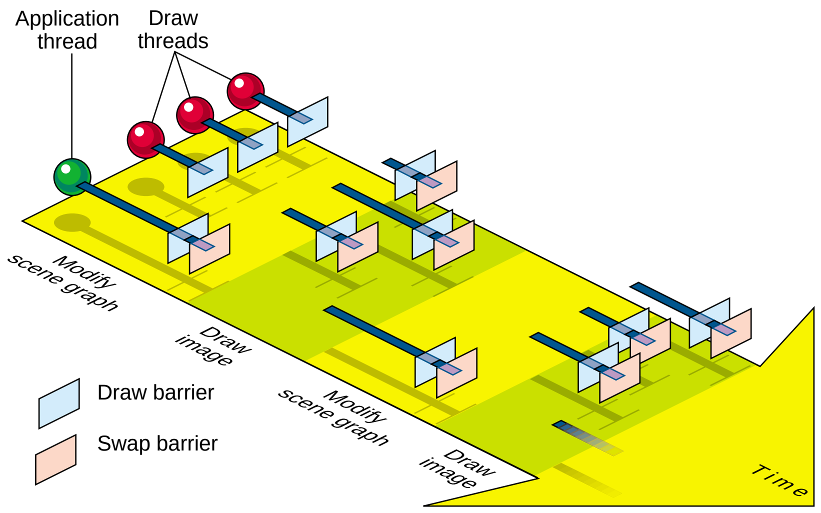

Figure 13-2 Blocking Action of Multiple Threads

The general order of events displayed in Figure 13-2 is as follows:

1. The application and draw threads are created and initialized.
2. The draw threads try to render but are stopped.
3. The application thread modifies the scene graph.
4. The scene graph is cleaned using **csField::cleanFields()**.
5. The application thread enters the Draw Barrier (semaphore) when it is ready to render the scenes.
6. The application thread tries to run but is stopped by the Swap Barrier until all draw threads have run.
7. The remainder of the draw threads run.
8. The draw threads enter the Swap Barrier.
9. when all threads enter the Swap Barrier they are released. The draw threads immediately enter the Draw Barrier and block.
10. The application thread modifies the scene graph.
11. The procedure repeats itself starting again at step 4.

Cleaning the csContext Fields

Before rendering, the application thread must call **csField::cleanFields()** to clean the **csContext** fields. This method forces the evaluation of all fields that have been dirty, thereby circumventing the normal lazy evaluation of field values. Generally, in Cosmo 3D, fields are updated only when queried.

To use this method, tracking of dirty fields must be enabled. This method enables safe traversal of a scene by parallel rendering threads; lazy evaluation is not thread-safe.

To clean the fields, you must:

1. Enable cleaning by using the following method:

   ```
   csField::enableDirtyFieldTracking();
   ```

2. Clean the fields by calling the following method each frame:

   ```
   csField::cleanFields();
   ```

To see if dirty field is enabled, use the following method:

```
csField::isDirtyFieldTrackingEnabled()
```

Multithreaded Example

For an example of a multi-threaded application, see *twothread.cxx* in *.../optimizer1.1/demos/optimizer/misc/.*

Chapter 14

Optimizing Rendering

One of the greatest challenges developers face is optimizing application performance. This chapter describes the Cosmo 3D nodes and programming techniques that can help optimize your application's performance.

The more vertices calculated and rendered, the slower the application's performance. If you can reduce calculations and rendering, you can improve application performance.

The following sections list means by which you can reduce the number of calculations made and the number of vertices rendered:

- "Face Culling" on page 152.
- "Back Patch Culling" on page 152.
- "Level of Detail Reduced for Performance" on page 160.
- "Culling the View Frustum" on page 160.
- "Performance Programming Techniques" on page 163.

Note: For more information about performance tools, see the *OpenGL Optimizer Programming Guide*.

Face Culling

When solid, three-dimensional geometry is rendered, the side of it facing away from the camera is normally hidden by the side that faces the camera. For example, when a sphere is rendered, you normally only see its front side.

You can avoid rendering the back side of a geometry using the **setCullFace()** method, defined in **csContext** and **csGeoSet** as follows:

```
void setCullFace(csContext::CullFaceEnum cullFace);
```

The argument in **setCullFace()** specifies how much of a geometry is rendered. The possible argument values, enumerated in **csContext::CullFaceEnum()**, include

- NO_CULL—Both front and back sides of geometries are rendered.
- FRONT_CULL—Only the back sides of all geometries are rendered.
- BACK_CULL—Only the front sides of all geometries are rendered.
- BOTH_CULL—Geometries are not rendered.

getCullFace() returns one of these values, whichever is current.

Not rendering either the front or back side of a geometry improves rendering performance.

Back Patch Culling

Back patch culling, like back face culling, eliminates from the rendering process parts of a geometry. Whereas back face culling is based on triangles, back patch culling is based on primitives, such as a tristrip or trifan. If, for example, you want to cull back faces:

- Back face culling prevents all triangles on the back side of a geometry from being rendered.
- Back patch culling prevents all primitives wholly on the back of the geometry from being rendered.

  If any part of a primitive is on the front of the geometry, the entire primitive is rendered, regardless of what part is on the front or back of the geometry.

To optimize the performance of your application, you would commonly back patch cull and then back face cull your scene graph.

Figure 14-1 shows the same geometry before and after back patch culling.

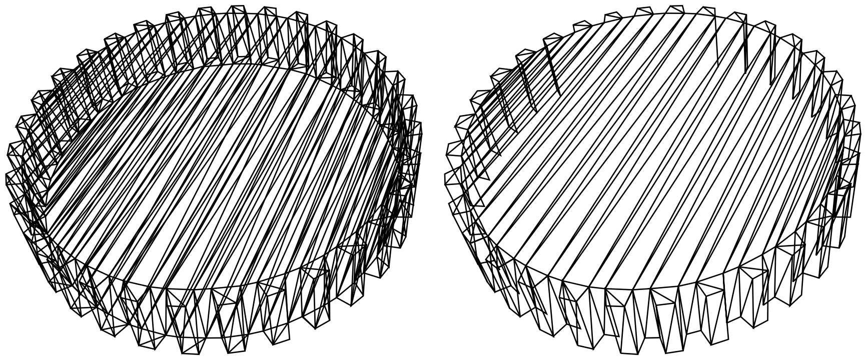

Figure 14-1 Before and After Back Patch Culling

Back Patch Culling Advantage

Back patch culling occurs before the primitives are processed by the graphics pipeline. As a result, back patch culling off-loads some of the graphics pipeline work to the CPU, yielding an added degree of parallelism between the two processors.

In back face culling, all of the triangles in a geometry are processed and then those on one side of a geometry are discarded. The amount of processing can be significant, including

1. Sending vertex data across the bus.
2. Transforming the vertices from object coordinates to clip coordinates.
3. Clipping to the viewing frustum.

Not displaying back faces is only a small part of the face processing. Consequently, culling back faces may not significantly enhance application performance.

In back patch culling, all primitives seen only on one side of a geometry are culled *before* their triangles are processed by the graphics pipeline. Back patch culling often improves application performance by not performing unnecessary processing.

Back patch culling usually reduces the work done by the graphics hardware but it does increase the workload of the host. Performance is enhanced when the time spent performing back patch culling is roughly equal to the time spent processing culled triangles in the graphics hardware.

When to Use Back Patch Culling

Back patch culling is most effective when:

- The primitives are composed of many elements.
- Most primitives are not on the front and back sides of geometric shapes.

If the primitives are short, the processing time of the back patch culling might approximate the rendering time for no net gain in performance. If most of the primitives wrap around to both sides of a geometry, few primitives are back patch culled.

Method of Calculation

To determine which side of a geometry an element of a primitive is on, the angle between the viewing vector and the normal to each element of a primitive is calculated. If the viewing angle is less than 90 degrees, the element is on the front of the geometry, as shown in Figure 14-2.

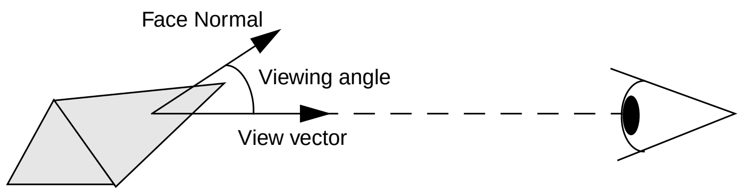

Figure 14-2 Viewing Angle

Updating the View Vector

As long as you use **csDrawAction::apply()** to initiate the back patch culling, you never need to calculate the view vector.

If you need greater control over the Draw process and use **csDrawAction::draw()**, you need to update the view vector using **csDrawAction::updateViewVector()**. **csDrawAction::getViewVector()** returns the view vector.

Normals

Two types of normals can be used to calculate the viewing angle:

- Face normal—the normal to the surface of an element.
- Primitive normal—the normal to all of the vertices of the elements when the PER_VERTEX_NORMAL binding is used, or the normal to the **csGeoSet** when any other binding is used.

Figure 14-3 shows these two types of normals.

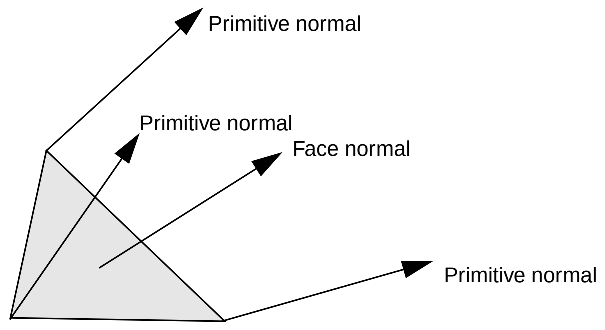

Figure 14-3 Face and Primitive Normals

Face normals can be calculated given the vertices; primitive normals must be provided by the scene graph.

The direction of the face normal follows the right hand rule: your thumb points in the direction of the normal when the fingers on your right hand wrap in the direction of the ordered vertices, as shown in Figure 14-4.

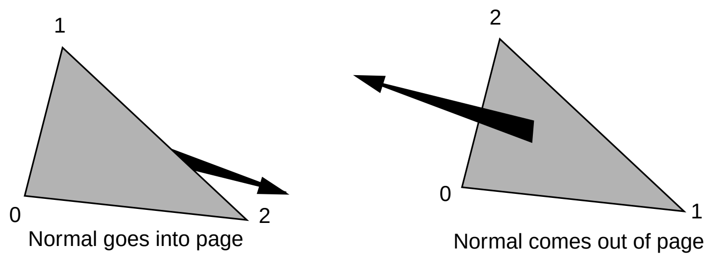

Figure 14-4 Direction of Normals

Choosing the Type of Normal

The **csGeoSet** method, **setBPCullVertNormModeEnable()**, sets the type of normal used to calculate the viewing angle: if the argument to the method evaluates TRUE, primitive normals are used, if the argument evaluates FALSE, face normals are used.

There is often a performance hit associated with face normals because they are calculated whereas primitive normals are provided by the scene graph. This performance hit occurs only when the back patch culling data is dynamically rebuilt, as explained in "Building Back Patch Culling Data for a csGeoSet" on page 158. The data is not rebuilt unless the primitive's coordinates or normals are changed.

csGeoSet::getBPCullPrimNormModeEnable() returns TRUE or FALSE, depending on whether the mode is primitive normals or face normals, respectively.

Primitive normals are used by default with the following primitive types:

- **csLineSet**
- **csLineStripSet**
- **csPointSet**

Face normals are used by default with the following primitive types:

- **csPolySet**
- **csQuadSet**
- **csTriFanSet**
- **csTriSet**
- **csTriStripSet**

If normals are not provided for a **csGeoSet** and the primitive normal mode is enabled, the **csGeoSet** is never back patch culled.

Using Back Patch Culling

There are two steps you take to implement back patch culling:

- Enable back patch culling.
- Build back patch culling data.

Enabling Back Patch Culling

You enable back patch culling by setting **csDrawAction**::**setBPCullMode()** with one of the following arguments:

- NONE—Disables back patch culling.
- DRAW_FRONT_FACING—Culls all primitives wholly on the back side of a geometry.
- DRAW_BACK_FACING—Culls all primitives wholly on the front side of a geometry.

csDrawAction::getBPCullMode() returns the back patch culling mode.

Once back patch culling is set, it is carried out whenever an **apply()** method is invoked with a **csDrawAction**.

Building Back Patch Culling Data for a csGeoSet

Before calling a **csDrawAction**, which triggers back patch culling, you must first build back patch culling data for the **csGeoSet**s of the scene. You only need to call **csGeoSet::buildBPCullData()** once; afterwards, the data can be automatically recomputed.

If back patch data does not exist for a **csGeoSet**, nothing is culled by back patch culling. You can check to see if back patch data exists for a **csGeoSet** by using **csGeoSet::existsBPCullData()**.

Back patch data is written to, and read from *.csb* files.

You can delete back patch data using **csGeoSet::deleteBPCullData()**.

Building Back Patch Culling Data for a Scene Graph

csGeoSet::buildBPCullData() builds the back patch culling data for a **csGeoSet**. It is the job of the application, however, to recursively go down through the scene graph and build back patch culling data for all of the **csGeoSet** nodes in a scene graph.

Updating Back Patch Culling Data

As the coordinates or the normals of primitives are changed, whether or not a primitive should be culled might also change. Optimizer, by default, automatically updates back patch culling data and culls the primitives correctly.

You can, however, turn off this automatic updating by setting **csGeoSet::setBPCullDynamicBuildMode()** to FALSE. Setting the argument to TRUE enables the automatic updating of back patch culling data.

csGeoSet::getBPCullDynamicBuildMode() returns TRUE if back patch culling data is automatically recomputed when **csGeoSets** change their coordinates or normals.

Back Patch Culling Code

Now that you understand all the facets of back patch culling, Example 14-1 presents the series of calls your application must make to implement back patch culling.

Example 14-1 Implementing Back Face Culling

```
// Create a draw action.
csDrawAction *drawAction = new csDrawAction;
...

// Set the back patch culling mode.
drawAction->setBPCullMode(csDrawAction::DRAW_FRONT_FACING);

// Build scene graph.
csNode *scene = new csNode;
...

// Build back patch culling data for scene graph by using
// csGeoSet::buildBPCullData() in the following developer-supplied
// method.
buildBPCullSceneData(scene);

// Apply the draw action to the scene graph.
drawAction->apply(scene);
```

Culling the View Frustum

View frustum culling eliminates from the rendering list all of those shapes not in the viewing frustum.

View frustum culling works best if the objects in a **csGroup** node are close together, for example, all of the nodes representing a body are linearly hierarchical. When this is the case, the CULL process only needs to visit the top of the body subgraph. If the body nodes were distributed horizontally, the CULL process would have to visit at least some of the other body nodes.

View frustum culling also works best when the **csShapes** are small compared to the full database size.

Objects that are roughly the same length in each of the three dimensions cull better than long, thin objects. An object that spans the database, for example, a beam across the ceiling of the building, cannot be culled as easily as two halves of the beam. It may be useful to divide up objects that can be easily divided.

OpenGL Optimizer provides tools to group together in the scene graph nodes whose shapes close together in world space.

Level of Detail Reduced for Performance

The children of a level of detail (**csLOD**) node each encapsulate a shape at a different level of detail. The factor of resolution between children of a **csLOD** is often one quarter; so when a lower resolution child replaces the current **csLOD** child displayed, only one quarter of the current number of vertices need to be rendered. The maximum reduction of detail is when all of the vertices of the highest-resolution image are reduced to a single pixel.

The **csLOD** (level of detail) node is a subclass of **csSwitch**. **csLOD** switches between its children nodes based on the proximity of an object to the camera.The further a shape is from the viewer, the less resolution needed to display it. Cosmo switches between the children automatically, based on range, to display a shape at the correct level of detail.

csLOD allows you to reach a compromise between performance and the level of detail rendered. For high quality images, a shape close to the camera should be rendered in high detail. When a shape recedes from the camera, the same level of detail is not necessary. Reducing the level of image detail reduces the number of vertices required to render a shape, which results in improved performance.

OpenGL Optimizer can create the **csLOD** child nodes.

Choosing a Child Node Based on Range

The distance, called the range, that determines which child of the **csLOD** is displayed is defined as the distance between a camera and a shape's center. Each child node of a **csLOD** node is associated with a range of distance values. The range is computed during the traversal of the scene graph. You set the range value using **csLOD** methods:

```
void setCenter(const csVec3f& c);
void getCenter(csVec3f &c) const;

void setRange(int index, float nearDistance,float farDistance);
void setRangeNear(int child,float distance);
void setRangeFar(int child,float distance);

int getNumRanges() const {return numRanges; }
float getRangeNear(int child) const;
float getRangeFar(int child) const;
```

The **setCenter()** method specifies the center of the LOD. The center point aids in calculating the range between the camera and the shape.

The **setRange()** method specifies the ranges over which a child node of the **csLOD** node is selected for display. The number of ranges must correspond to the number of **csLOD** child nodes. If that is not the case:

- If too few ranges are specified, the highest-order child nodes are ignored.
- If too many ranges are specified, the extra ranges are ignored.

Instead of using **setRange()**, you can use **setRangeNear()** together with **setRangeFar()** to specify the range over which a child node of a **csLOD** node is selected for display.

The camera may disregard range values and

- Display an already-fetched level of detail while a higher level of detail is downloaded from disk.
- Adjust the level of detail displayed to maintain a constant frame rate; this is always the case if you leave the **range()** field empty.
- Disregard the range values for any other implementation-dependent reason.

Tip: For best results, specify ranges only where necessary; give browsers as much freedom as possible to choose levels of detail based on performance.

Transitioning Between Levels of Detail

The **transition()** method specifies the range over which one **csLOD** child changes into the next, as shown in Figure 14-5.

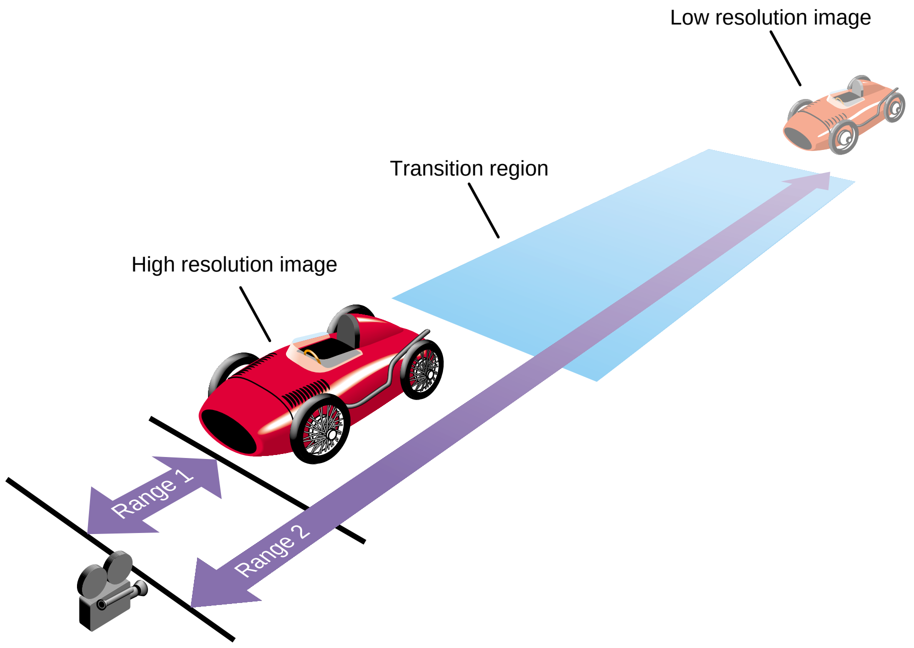

Figure 14-5 csLOD Ranges

Performance Programming Techniques

The following sections provide programming tips for improving the performance of your application:

- "Minimize Use of csAppearance Fields" on page 163.
- "Minimize Use of csAppearance Modes" on page 163.
- "Indexing csGeoSet Attributes" on page 164.
- "Setting the Transformation Matrix Directly" on page 164.
- "Compiling Part of a Scene Graph" on page 164.

Minimize Use of csAppearance Fields

Many of the fields in **csContext** set the appearance of a geometry. The fields in **csAppearance** match those in **csContext**. Setting a **csAppearance** field overrides the values of the same fields set in **csContext**. Overriding **csContext** values, however, reduces performance because the field has to be reevaluated every time the **csAppearance** object is applied.

To maximize performance:

- Set **csContext** fields to values that satisfy a majority of the shapes in a scene.
- Set the inherit field in **csAppearance** to inherit, not apply, those fields.

In this way, you set the minimum number of **csAppearance** fields.

Minimize Use of csAppearance Modes

Some of the fields set in **csAppearance** are much more graphics-intensive than others. In particular, the blending, texture, and lighting fields require larger amounts of CPU time. To improve performance, it is better not to turn on these modes if your application does not need them.

Indexing csGeoSet Attributes

You can specify the appearance of all the **csGeoSet** elements making up a geometry either individually or collectively. You have the option of specifying the attribute values sequentially, so that the first element is described by the first **csAttribute** values or you can use an index system.

Choosing to index the attribute values (or not) can dramatically affect application performance. The general rule to remember when indexing or not is to determine whether many elements share the same vertices, or not. If many elements share the same vertex, index the attribute values; if a vertex is not shared by many elements, specify the attribute values of the elements sequentially.

For more information about indexing, see “Indexing Attributes” on page 23.

Setting the Transformation Matrix Directly

Whether you set the transformation matrix explicitly or you use the **csContext** methods that set the transformation matrix for you, rendering performance is optimized for the following reason: a shape can be translated, scaled, and rotated. Rather than computing these methods every time a shape is drawn, the transformation matrix represents the product of all three methods. Likewise, when a transformation matrix node is the parent of many shape nodes, the transformation for all of the children shape nodes is captured in a single transformation matrix.

Compiling Part of a Scene Graph

Although there are no restrictions on the way in which you create a scene graph, it is customary to find that pieces deep in a scene graph branch add to pieces above them, which add to pieces above them until an entire object is described. For example, the lowest node in a branch might be a toe, the node above it might be the foot, the node above that the leg, the node above that the body; taken together, the nodes describe one side of the lower half of a body, as shown in Figure 14-6.

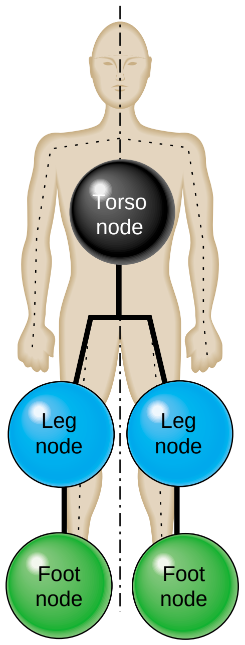

Figure 14-6 Arranging Scene Graph Nodes

When an action traverses a scene graph, the more nodes it visits, the longer it takes to execute the action. If scene graph branches are deep, the traversal can become expensive. To correct this problem, if you find that the elements in a branch do not change often, you can precompile that branch using **csCompileAction**. This action compiles a specified subgraph into a data structure, which optimizes setting the traversal state.

To compile a subgraph, create a **csCompileAction** object and apply it to the root node of the scene graph, as follows

```
csCompileAction *compileIt = new csCompileAction;
compileIt->apply(node_name);
```

node_name is the name of the node below which you want to precompile.

Chapter 15

Adding Sounds To Virtual Worlds

You can incorporate sound into your virtual worlds by including at least one **csSound** node in a scene graph and by invoking a **csSoundAction**. The **csSoundAction** plays the sound file specified in the **csSound** node. This node also includes parameters, such as volume, for playing the sound.

This chapter describes how to set and play sound using Cosmo 3D.

These are the sections in this chapter:

Overview

A **csSound** node contains the location of a sound file and the parameters used for playing it. To play a sound file in a virtual world, attach one or more **csSound** objects to a scene graph and apply a **csSoundAction** to it. To associate a specific sound to a specific shape, make the **csSound** object and shape nodes children of the same group nodes.

A **csSound** node references a **csAudioClip** node and a **csAudioClip** node references a **csAudioSamples** node, as shown in Figure 15-1.

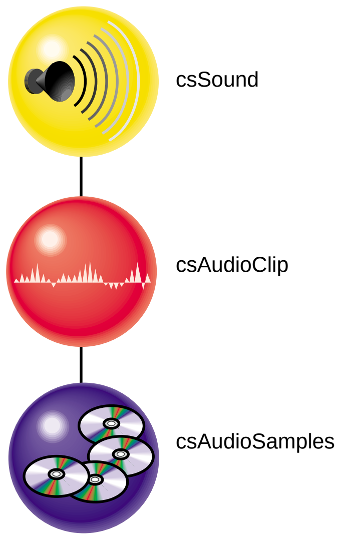

Figure 15-1 Sound Classes

A **csAudioSamples** node contains the raw sound data. A **csAudioClip** node specifies how a sound file should be played.

For more information about **csAudioClip**, see "How to Play a Sound File" on page 173. For more information about **csAudioSamples**, see "Specifying Audio Files" on page 174.

csSound Fields

The fields in **csSound** specify the sound source to play by specifying a **csAudioClip** object. **csSound** can optionally specify the location, the direction of the sound, and the spatial characteristics of the sound.

```
void                setSource(csAudioClip* audioClip);
void                setSpatialize(csBool spatialize);

void                setControl(ControlEnum control);

void                setLocation(const csVec3f& location);
void                setLocation(csFloat v0, csFloat v1, csFloat v2);
void                setDirection(const csVec3f& direction);
void                setDirection(csFloat v0, csFloat v1, csFloat v2);

void                setIntensity(csFloat intensity);
void                setMaxIntensity(csFloat maxIntensity);
void                setCullIntensity(csFloat cullIntensity);

void                setCurrentFrame(csFloat currentFrame);
void                setPriority(csFloat priority);

void                setMinFront(csFloat minFront);
void                setMaxFront(csFloat maxFront);
void                setMinBack(csFloat minBack);
void                setMaxBack(csFloat maxBack);
const csIntArray&   getEvents();
```

The following sections describe these fields.

Choosing Sound Samples to Play

To specify how to play a sound file, pass a **csAudioClips** object to **setSource().** The **csAudioClips** object identifies a **csAudioSample** object, which contains the sound file to play. For more information about **csAudioClips**, see “How to Play a Sound File” on page 173.

A sound file commonly contains more than one sound sample. Some samples may contain more than one sound channel per sound interval to create, for example, stereo.

To choose a starting location in a sound file, pass the starting frame to the **setCurrentFrame()** field. A frame is equal to (1/*SampleRate)* of a second. The sample rate might be, for example, 44KHz, or 44,000 Hz. If, for example, you pass 44000.0 into the **setCurrentFrame()** field, the sound would begin playing one second ($44000 \times 1/44000 = 1$) into the sound file.

Sound Priority

Your application can play only a limited number of sounds at the same time. The factors that determine whether or not a sound is heard include

- The proximity of the listener to the source.
- The priority level of the sound.

Higher priority sounds are heard instead of lower priority sounds if too many sounds could possibly be heard by the listener at the same time. Set the priority level of a sound in the **setPriority()** method.

Playing the Sound File

The **setControl()** field provides an intuitive interface for playing the sound sample in sound files. You pass into **setControl()** any of the ControlEnum values, including PLAY, PAUSE, REWIND, FASTFORWARD, and STOP.

Locating and Directing the Sound

To enable all of the other effects implemented by the fields in the **csSound** node, covered in the next section, pass a non NULL value to **setSpatialize()**. If you pass a NULL value to the field, the volume is a constant value throughout the scene. This choice is appropriate, for example, for background music.

To locate the sound source in a scene, pass its coordinates to the **setLocation()** field.

Cosmo 3D gives you a great deal of control over how sound propagates from the source. When you supply a vector describing the direction of the sound, the sound propagates in all directions, but attenuates least in specified direction. The attenuation of the sound over distance is characterized by an ellipse, as shown in Figure 15-2.

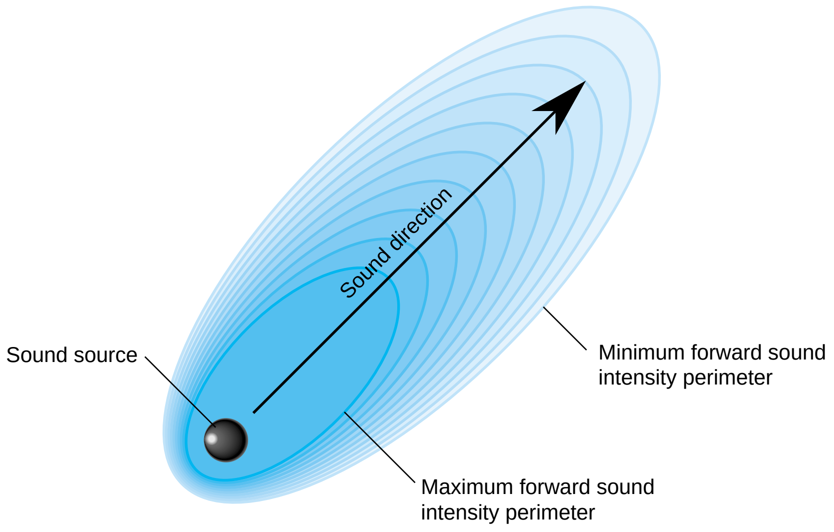
Sound direction
Sound source
Minimum forward sound intensity perimeter
Maximum forward sound intensity perimeter

Figure 15-2 Sound Direction

In this figure, (1.0, 1.0, 0.0) is passed as the direction vector to **setDirection()**. The ellipse tips, accordingly, at a 45 degree angle.

Note: A transformation node can reorient the sound's location and direction.

Cosmo 3D provides the following limiting tools to fashion the attenuation of the sound over the ellipse:

- Maximum intensity—defines the maximum possible volume regardless of how close the listener is to the sound source.
- Minimum intensity—defines the lowest possible volume of a sound. In practice, since this value is often set to zero, the minimum intensity perimeter defines the range of the sound.

The **setIntensity()** field specifies the volume of the sound at its source. The **setMaxIntensity()** field specifies the maximum volume of a sound. If a maximum intensity is set, as is the case in Figure 15-2, the intensity of the sound within the maximum intensity perimeter does not attenuate and is equal to the volume specified by **setIntensity()**. The **maxFront()** field specifies the maximum intensity perimeter within which the listener hears the maximum volume of the sound.

Outside of the maximum intensity perimeter, the intensity of the sound attenuates over distance until it reaches the minimum intensity perimeter. Beyond the minimum intensity perimeter, the volume of the sound source is constant, defined by the **setCullIntensity()** field.

Reverse Direction Sound

Cosmo 3D also provides a complimentary set of fields that allow you to define the propagation and attenuation of the same sound in the opposite direction, as shown in Figure 15-3.

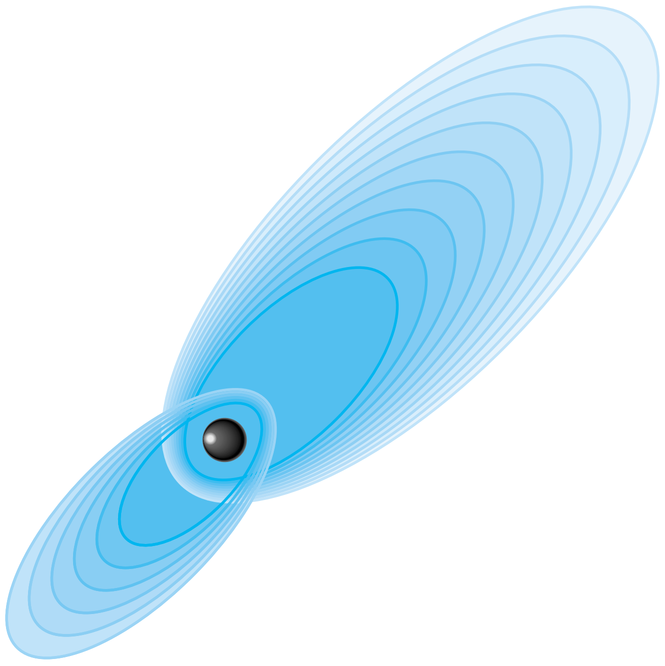

Figure 15-3 Forward and Reverse Sound Propagation

The minimum and maximum intensities in the reverse direction are the same as those in the forward direction. The minimum and maximum intensity perimeters, however, are specified separately with the **minBack()**, and **maxBack()** fields.

How to Play a Sound File

You use **csAudioClip** to specify how to play the sound files referenced in the **csAudioSamples** node.

csAudioClip contains the following fields:

```
csMFString* url() const;
void setSamples(csAudioSamples* samples);
void setPitch(csFloat pitch);
void setStartTime(csTime startTime);
void setStopTime(csTime stopTime);
void setDoppler(csBool doppler);
void setLoop(csBool loop);
void setDescription(const csString& description);
csTime getDuration();
csBool getIsActive();
```

These **set()** fields have corresponding **get()** fields. Table 15-1 describes how these fields are used.

Table 15-1 csAudioClip Fields

Field	Description
url	Specifies a WWW URL where the sound source file can be found.
setSamples	Attaches the csAudioClip object to a csAudioSamples object.
setPitch	Adjusts the pitch of a sound sample.
setStartTime	Specifies a beginning time for the sound sample to begin playing. Time here is an expression of clock time.
setStopTime	Specifies an ending time for the sound sample to stop playing. Time here is an expression of clock time.
setDoppler	Enables the doppler effect, which is the attenuation of a sounds pitch based on the velocity of the sound source relative to the listener, for example, when a sound source, like a train whistle, approaches a listener rapidly, the pitch sounds higher; when the same sound source passes the listener, the pitch lowers.
setLoop	Allows a sound file to keep playing.
setDescription	Provides a description in the node of the sound source.

Table 15-1 (continued) csAudioClip Fields

Field	Description
getDuration	Returns the duration of the playing of the sound source; subtracts setStartTime from setStopTime.
getIsActive	Returns whether or not the sound should be played.

Example 15-1 sets all of the fields in a **csAudioSamples** node.

Example 15-1 Setting the Fields in a csAudioClip Object

```
// create a csAudioClip object
csAudioClip* clip = new csAudioClip();

// attach the audio clip to a csAudioSamples object
clip->setSamples(csAudioSamples truck_horn);

// Set the parameters of the csAudioClip object
clip->setPitch(1.0);
clip->setIntensity(1.0);
clip->setMaxIntensity(4.0);
clip->setLoop(TRUE);
clip->setDescription("Truck horn");
```

Specifying Audio Files

You use the **csAudioSamples** node to specify the source of the audio files and a variety of parameters that describe those sound samples. To use the audio files specified by this node, pass a **csAudioSamples** object as the argument to **csAudioClip::setSamples()**, for example,

```
csAudioSamples* truck_horn_file = new csAudioSamples;
csAudioClip* truck_horn_style = new csAudioClip;

truck_horn_style->setSamples(truck_horn_file);
```

In this example, the *truck_horn_file* is used as the audio file for the *truck_horn_style* object.

csAudioSamples has the following fields:

```
void                setFileName(const csString& fileName);
void                setNumFrames(csInt numFrames);
void                setSampleRate(csFloat sampleRate);
void                setSampleSize(csInt sampleSize);
void                setSampleType(SampleTypeEnum sampleType);
void                setNumChannels(csInt numChannels);
void                setSampleScale(csFloat scale);
void                setLoadStatus(LoadStatusEnum loadStatus);
LoadStatusEnum      getLoadStatus();
csMFByte*           samples() const;
```

These **set()** fields have corresponding **get()** fields. Table 15-2 describes how these fields are used.

Table 15-2 Fields of csSoundSamples

Field	Description
setFileName	Attaches the csAudioSample node to a specific sound source file.
setNumFrames	Is the equivalent to the sampling rate of the sound sample, for example, 44 KHz.
setSampleRate	Specifies the sampling rate of the sound sample.
setSampleSize	Specifies the size of the source sound files.
setSampleType	Specifies the format of the sampling rate. Valid values include UNSIGNED_INT_SAMPLE_TYPE or FLOAT_SAMPLE_TYPE.
setNumChannels	Specifies the number of channels for each sound, for example, stereo has two channels, quad sound has four channels.
setSampleScale	Scales the overall volume of the sound sample. If if sound sample was recorded to loud or soft compared to other sound samples, you can scale the volume of the sound file so that its volume matches that of the other sound files.
setLoadStatus	Returns the status of whether or not the sound file loaded; valid values include LOAD_NEEDED, LOAD_FAILED, LOAD_PENDING, and LOAD_COMPLETE.
samples	Returns the multivalued array field that contains the actual audio samples.

Manipulating the Audio Samples Directly

csAudioSample::samples() returns the multivalued array field that contains the actual audio samples. This handle allows you to directly manipulate the array field. For example, to set a sound value, use the following code:

```
samples->set(index, value);
```

To set the number of samples and then edit the array directly, use the following code:

```
samples->setCount(16*44400);
char *samps = samples->edit();
for(i=0;i<16*44400);i++)
    samps[i]=DETERMINE_SAMPLE(i);
samples->editDone();
```

Example Setting a csAudioSamples Node

Example 15-2 sets all of the fields in a **csAudioSamples** node.

Example 15-2 Setting the Fields in an csAudioSamples Object

```
// create an audio sample node
csAudioSamples* horn = new csAudioSamples;

// attach the audio sample node to a specific file
horn->setFileName(“truck_horn.xxx”);

// Set the parameters for the source sound file
horn->setNumFrames(44000.0);
horn->setSampleRate(44000.0);
horn->setSampleSize(2000);
horn->setSampleType(UNSIGNED_INT_SAMPLE_TYPE);
horn->setNumChannels(2);
horn->setSampleScale(1.0);
...

// Load the sound sample by setting the audiosample filename
horn->load();

// Make sure the sound loaded successfully
if (horn->getLoadStatus() == LOAD_FAILED)
    abort();
```

When **load()** is called, Cosmo 3D reads the samples in the file into the sample field directly.

Playing Sound in Immediate Mode

When a **csSoundAction** is invoked on a scene graph, the action traverses the scene graph and gathers a list of active **csSound** nodes. The action notifies **csContext** internally of this list of nodes. When the context is applied to the rendering pipeline, the sounds specified in the associated **csAudioSamples** nodes are played.

You can also play a sound file immediately. Instead of using a **csSoundAction** to trigger the playing of the sound file, you use a **csSoundPlayer** node.

All of the code used to play sounds, either using **csSoundAction** or **csSoundPlayer**, is encapsulated in **csSoundPlayer**. Consequently, all of the field settings discussed previously in this chapter also need to be specified in **csSoundPlayer**.

csSoundPlayer Methods

csSoundPlayer methods provide sophisticated control over the sound source and the “microphone” recording the sound. Some of those controls include:

- Position of the sound source and microphone.
- Motion of the sound source and microphone; helpful in simulating a Doppler shift.
- Scaling of the sound source intensity and frequency.
- Number of recording channels.

Chapter 15

Adding Sounds To Virtual Worlds

You can incorporate sound into your virtual worlds by including at least one **csSound** node in a scene graph and by invoking a **csSoundAction**. The **csSoundAction** plays the sound file specified in the **csSound** node. This node also includes parameters, such as volume, for playing the sound.

This chapter describes how to set and play sound using Cosmo 3D.

These are the sections in this chapter:

Overview

A **csSound** node contains the location of a sound file and the parameters used for playing it. To play a sound file in a virtual world, attach one or more **csSound** objects to a scene graph and apply a **csSoundAction** to it. To associate a specific sound to a specific shape, make the **csSound** object and shape nodes children of the same group nodes.

A **csSound** node references a **csAudioClip** node and a **csAudioClip** node references a **csAudioSamples** node, as shown in Figure 15-1.

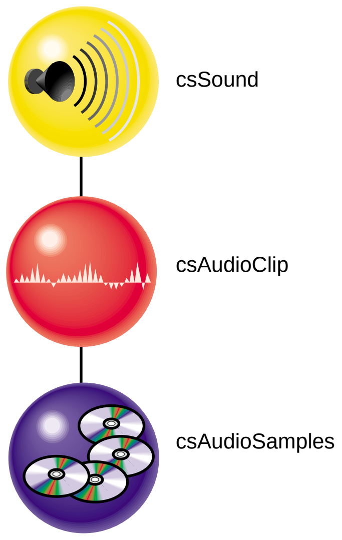

Figure 15-1 Sound Classes

A **csAudioSamples** node contains the raw sound data. A **csAudioClip** node specifies how a sound file should be played.

For more information about **csAudioClip**, see “How to Play a Sound File” on page 173. For more information about **csAudioSamples**, see “Specifying Audio Files” on page 174.

csSound Fields

The fields in **csSound** specify the sound source to play by specifying a **csAudioClip** object. **csSound** can optionally specify the location, the direction of the sound, and the spatial characteristics of the sound.

```
void                setSource(csAudioClip* audioClip);
void                setSpatialize(csBool spatialize);

void                setControl(ControlEnum control);

void                setLocation(const csVec3f& location);
void                setLocation(csFloat v0, csFloat v1, csFloat v2);
void                setDirection(const csVec3f& direction);
void                setDirection(csFloat v0, csFloat v1, csFloat v2);

void                setIntensity(csFloat intensity);
void                setMaxIntensity(csFloat maxIntensity);
void                setCullIntensity(csFloat cullIntensity);

void                setCurrentFrame(csFloat currentFrame);
void                setPriority(csFloat priority);

void                setMinFront(csFloat minFront);
void                setMaxFront(csFloat maxFront);
void                setMinBack(csFloat minBack);
void                setMaxBack(csFloat maxBack);
const csIntArray&   getEvents();
```

The following sections describe these fields.

Choosing Sound Samples to Play

To specify how to play a sound file, pass a **csAudioClips** object to **setSource().** The **csAudioClips** object identifies a **csAudioSample** object, which contains the sound file to play. For more information about **csAudioClips**, see “How to Play a Sound File” on page 173.

A sound file commonly contains more than one sound sample. Some samples may contain more than one sound channel per sound interval to create, for example, stereo.

To choose a starting location in a sound file, pass the starting frame to the **setCurrentFrame()** field. A frame is equal to (1/*SampleRate)* of a second. The sample rate might be, for example, 44KHz, or 44,000 Hz. If, for example, you pass 44000.0 into the **setCurrentFrame()** field, the sound would begin playing one second ($44000 \times 1/44000 = 1$) into the sound file.

Sound Priority

Your application can play only a limited number of sounds at the same time. The factors that determine whether or not a sound is heard include

- The proximity of the listener to the source.
- The priority level of the sound.

Higher priority sounds are heard instead of lower priority sounds if too many sounds could possibly be heard by the listener at the same time. Set the priority level of a sound in the **setPriority()** method.

Playing the Sound File

The **setControl()** field provides an intuitive interface for playing the sound sample in sound files. You pass into **setControl()** any of the ControlEnum values, including PLAY, PAUSE, REWIND, FASTFORWARD, and STOP.

Locating and Directing the Sound

To enable all of the other effects implemented by the fields in the **csSound** node, covered in the next section, pass a non NULL value to **setSpatialize()**. If you pass a NULL value to the field, the volume is a constant value throughout the scene. This choice is appropriate, for example, for background music.

To locate the sound source in a scene, pass its coordinates to the **setLocation()** field.

Cosmo 3D gives you a great deal of control over how sound propagates from the source. When you supply a vector describing the direction of the sound, the sound propagates in all directions, but attenuates least in specified direction. The attenuation of the sound over distance is characterized by an ellipse, as shown in Figure 15-2.

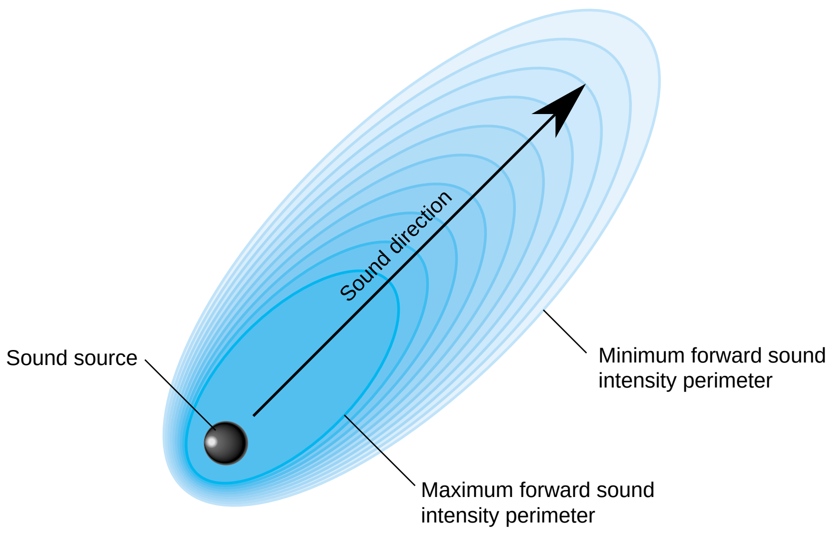
Sound direction
Sound source
Minimum forward sound intensity perimeter
Maximum forward sound intensity perimeter

Figure 15-2 Sound Direction

In this figure, (1.0, 1.0, 0.0) is passed as the direction vector to **setDirection()**. The ellipse tips, accordingly, at a 45 degree angle.

Note: A transformation node can reorient the sound's location and direction.

Cosmo 3D provides the following limiting tools to fashion the attenuation of the sound over the ellipse:

- Maximum intensity—defines the maximum possible volume regardless of how close the listener is to the sound source.
- Minimum intensity—defines the lowest possible volume of a sound. In practice, since this value is often set to zero, the minimum intensity perimeter defines the range of the sound.

The **setIntensity()** field specifies the volume of the sound at its source. The **setMaxIntensity()** field specifies the maximum volume of a sound. If a maximum intensity is set, as is the case in Figure 15-2, the intensity of the sound within the maximum intensity perimeter does not attenuate and is equal to the volume specified by **setIntensity()**. The **maxFront()** field specifies the maximum intensity perimeter within which the listener hears the maximum volume of the sound.

Outside of the maximum intensity perimeter, the intensity of the sound attenuates over distance until it reaches the minimum intensity perimeter. Beyond the minimum intensity perimeter, the volume of the sound source is constant, defined by the **setCullIntensity()** field.

Reverse Direction Sound

Cosmo 3D also provides a complimentary set of fields that allow you to define the propagation and attenuation of the same sound in the opposite direction, as shown in Figure 15-3.

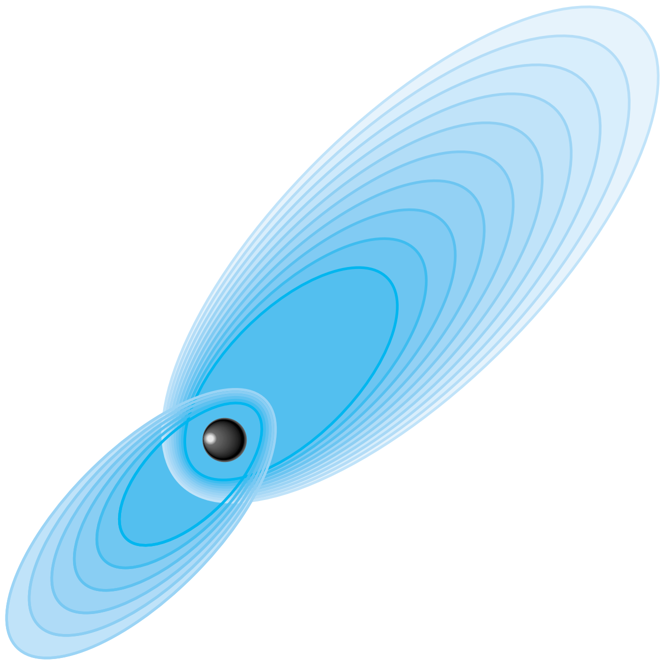

Figure 15-3 Forward and Reverse Sound Propagation

The minimum and maximum intensities in the reverse direction are the same as those in the forward direction. The minimum and maximum intensity perimeters, however, are specified separately with the **minBack()**, and **maxBack()** fields.

How to Play a Sound File

You use **csAudioClip** to specify how to play the sound files referenced in the **csAudioSamples** node.

csAudioClip contains the following fields:

```
csMFString* url() const;
void setSamples(csAudioSamples* samples);
void setPitch(csFloat pitch);
void setStartTime(csTime startTime);
void setStopTime(csTime stopTime);
void setDoppler(csBool doppler);
void setLoop(csBool loop);
void setDescription(const csString& description);
csTime getDuration();
csBool getIsActive();
```

These **set()** fields have corresponding **get()** fields. Table 15-1 describes how these fields are used.

Table 15-1 csAudioClip Fields

Field	Description
url	Specifies a WWW URL where the sound source file can be found.
setSamples	Attaches the csAudioClip object to a csAudioSamples object.
setPitch	Adjusts the pitch of a sound sample.
setStartTime	Specifies a beginning time for the sound sample to begin playing. Time here is an expression of clock time.
setStopTime	Specifies an ending time for the sound sample to stop playing. Time here is an expression of clock time.
setDoppler	Enables the doppler effect, which is the attenuation of a sounds pitch based on the velocity of the sound source relative to the listener, for example, when a sound source, like a train whistle, approaches a listener rapidly, the pitch sounds higher; when the same sound source passes the listener, the pitch lowers.
setLoop	Allows a sound file to keep playing.
setDescription	Provides a description in the node of the sound source.

Table 15-1 (continued) csAudioClip Fields

Field	Description
getDuration	Returns the duration of the playing of the sound source; subtracts setStartTime from setStopTime.
getIsActive	Returns whether or not the sound should be played.

Example 15-1 sets all of the fields in a **csAudioSamples** node.

Example 15-1 Setting the Fields in a csAudioClip Object

```
// create a csAudioClip object
csAudioClip* clip = new csAudioClip();

// attach the audio clip to a csAudioSamples object
clip->setSamples(csAudioSamples truck_horn);

// Set the parameters of the csAudioClip object
clip->setPitch(1.0);
clip->setIntensity(1.0);
clip->setMaxIntensity(4.0);
clip->setLoop(TRUE);
clip->setDescription(“Truck horn”);
```

Specifying Audio Files

You use the **csAudioSamples** node to specify the source of the audio files and a variety of parameters that describe those sound samples. To use the audio files specified by this node, pass a **csAudioSamples** object as the argument to **csAudioClip::setSamples()**, for example,

```
csAudioSamples* truck_horn_file = new csAudioSamples;
csAudioClip* truck_horn_style = new csAudioClip;

truck_horn_style->setSamples(truck_horn_file);
```

In this example, the *truck_horn_file* is used as the audio file for the *truck_horn_style* object.

csAudioSamples has the following fields:

```
void                setFileName(const csString& fileName);
void                setNumFrames(csInt numFrames);
void                setSampleRate(csFloat sampleRate);
void                setSampleSize(csInt sampleSize);
void                setSampleType(SampleTypeEnum sampleType);
void                setNumChannels(csInt numChannels);
void                setSampleScale(csFloat scale);
void                setLoadStatus(LoadStatusEnum loadStatus);
LoadStatusEnum      getLoadStatus();
csMFByte*           samples() const;
```

These **set()** fields have corresponding **get()** fields. Table 15-2 describes how these fields are used.

Table 15-2 Fields of csSoundSamples

Field	Description
setFileName	Attaches the csAudioSample node to a specific sound source file.
setNumFrames	Is the equivalent to the sampling rate of the sound sample, for example, 44 KHz.
setSampleRate	Specifies the sampling rate of the sound sample.
setSampleSize	Specifies the size of the source sound files.
setSampleType	Specifies the format of the sampling rate. Valid values include UNSIGNED_INT_SAMPLE_TYPE or FLOAT_SAMPLE_TYPE.
setNumChannels	Specifies the number of channels for each sound, for example, stereo has two channels, quad sound has four channels.
setSampleScale	Scales the overall volume of the sound sample. If if sound sample was recorded to loud or soft compared to other sound samples, you can scale the volume of the sound file so that its volume matches that of the other sound files.
setLoadStatus	Returns the status of whether or not the sound file loaded; valid values include LOAD_NEEDED, LOAD_FAILED, LOAD_PENDING, and LOAD_COMPLETE.
samples	Returns the multivalued array field that contains the actual audio samples.

Manipulating the Audio Samples Directly

csAudioSample::samples() returns the multivalued array field that contains the actual audio samples. This handle allows you to directly manipulate the array field. For example, to set a sound value, use the following code:

```
samples->set(index, value);
```

To set the number of samples and then edit the array directly, use the following code:

```
samples->setCount(16*44400);
char *samps = samples->edit();
for(i=0;i<16*44400);i++)
    samps[i]=DETERMINE_SAMPLE(i);
samples->editDone();
```

Example Setting a csAudioSamples Node

Example 15-2 sets all of the fields in a **csAudioSamples** node.

Example 15-2 Setting the Fields in an csAudioSamples Object

```
// create an audio sample node
csAudioSamples* horn = new csAudioSamples;

// attach the audio sample node to a specific file
horn->setFileName(“truck_horn.xxx”);

// Set the parameters for the source sound file
horn->setNumFrames(44000.0);
horn->setSampleRate(44000.0);
horn->setSampleSize(2000);
horn->setSampleType(UNSIGNED_INT_SAMPLE_TYPE);
horn->setNumChannels(2);
horn->setSampleScale(1.0);
...

// Load the sound sample by setting the audiosample filename
horn->load();

// Make sure the sound loaded successfully
if (horn->getLoadStatus() == LOAD_FAILED)
    abort();
```

When **load()** is called, Cosmo 3D reads the samples in the file into the sample field directly.

Playing Sound in Immediate Mode

When a **csSoundAction** is invoked on a scene graph, the action traverses the scene graph and gathers a list of active **csSound** nodes. The action notifies **csContext** internally of this list of nodes. When the context is applied to the rendering pipeline, the sounds specified in the associated **csAudioSamples** nodes are played.

You can also play a sound file immediately. Instead of using a **csSoundAction** to trigger the playing of the sound file, you use a **csSoundPlayer** node.

All of the code used to play sounds, either using **csSoundAction** or **csSoundPlayer**, is encapsulated in **csSoundPlayer**. Consequently, all of the field settings discussed previously in this chapter also need to be specified in **csSoundPlayer**.

csSoundPlayer Methods

csSoundPlayer methods provide sophisticated control over the sound source and the "microphone" recording the sound. Some of those controls include:

- Position of the sound source and microphone.
- Motion of the sound source and microphone; helpful in simulating a Doppler shift.
- Scaling of the sound source intensity and frequency.
- Number of recording channels.

Appendix A

Cosmo Basic Types

This chapter discusses all of the basic types that are used in other Cosmo 3D classes. The basic class types fall into the following categories:

- Array storage—stores data.
- Vector classes—stores vectors.
- Bounding shapes—creates a volume around a specified shape.
- Field classes—specifies the classes for the node fields: single value.
- Other math classes—miscellaneous math classes.

This chapter examines each of these class categories.

These are the sections in this chapter:

Array Storage Class Types

The array classes store data.

- csData—stores raw data.
- csArray—is a virtual array class.
- Array-derived classes—are derivations of csArray.

The following sections describe each of these array classes.

Data Class

The **csData** class is similar to malloc: it stores raw data. You can use the **csData** class directly, such as in storing data in arrays, but it is more common to derive your own class from it.

The methods in the class set and delete the amount of storage necessary for the data being stored:

```
void* getData () const;
int getSize () const;
void operator delete (void* ptr);
void* operator new (size_t nbytes);
```

getData() returns a pointer to the raw bytes of the **csData**.

getSize() returns the size of the data.

new() creates a new **csData** containing *nbytes* bytes.

delete() deallocates the object created by the **new()** method.

Array Classes

csArray is a virtual array class from which all other array classes are derived. Arrays are used as storage vehicles for a variety of types. Cosmo 3D provides a wealth of **csArray**-derived array classes for different types, including:

- csPtrArray—An array of pointers often used to point to values in other arrays.
- csFieldArray—An array of fields.
- csByteArray—An array of bytes.
- csIntArray—An array of integers.
- csFloatArray—An array of floats.
- csVec2fArray—An array of Vec2f classes.
- csVec3fArray—An array of Vec3f classes.
- csVec3sArray—An array of Vec3s classes.
- csVec4fArray—An array of Vec4f classes.
- csMatrix4fArray—An array of Matrix4f classes.
- csRotationArray—An array of rotation vectors.
- csStringArray—An array of strings.
- csShortArray—An array of shorts.
- csFieldInfoArray—An array of field descriptions.
- csRefArray—An array of pointers to containers.
- csEventArray—An array of events; an event is a user action, such as a mouse click.

The methods in all of the array classes are similar, as described in the following section.

Array Methods

The methods in **csArray** and all of the derived array classes are similar; the differences stem from the different types filling the arrays. The following example explains the methods in **csIntArray** but you can easily apply the same descriptions to all of the other array classes.

To fill an array, or to retrieve values from an array, use one of the **set()** or **get()** methods in the class, respectively, replacing <type> with the base type of the array.

```
void set(int i, csInt t);
void get(int i, csInt& t) const;
<type> get(int i) const;
void set(const csIntArray &l);
```

- *i* is the position of the value in the array.
- *t*is the value of that element in the array.

The second version of the **set()** method allows you to copy the contents of one array to another.

You can fill an array by setting groups of values using the following methods:

```
void        setRange(int i, int count, csInt vals[]);
void        getRange(int i, int count, csInt vals[]);
void        fillRange(int i, int count, csInt vals[]);
```

- *i* is the position in the array where you want to begin setting values.
- *count* is the number of array elements you want to set.
- *vals[]* is an array of values that you want to enter into the array.

The **fillRange()** method sets all of the values in an array to the value passed in the argument.

The operator, [], assigns values.

```
<type> operator[](int i) const;
```

The following methods manipulate the values in the array:

```
void        append(<type> elt);
void        insert(int index, <type> elt);
int         replace(csInt old, <type> elt);
int         remove(<type> elt);
void        removeIndex(int i);
int         fastRemove(<type> elt);
void        fastRemoveIndex(int i);
int         find(<type> elt) const;
void        write(csOutput *out);
```

- Use the **append()** method to add a value after the last value in an array.
- Use the **insert()** method to insert a value at a specified index location.
- Use the **replace()** method to replace one value with another.
- Use the **remove()** method to remove the first value it finds that matches its argument.
- Use the **removeIndex()** method to remove a specific value or a value located at a specific index.

You can also remove array elements more quickly: **fastRemove()** removes the array element, *elt*, and replaces it with the last element of the array, whereas **remove()** moves all the remaining elements down one. Also, you can remove the array element, *elt*, by passing in the index value, *i*, to **fastRemoveIndex()**.

You can find the index of a specific value in an array using the **find()** method. You can also write the contents of the array to the **csOutput** passed using the **write()** method.

Returning Array Data

To return a pointer directly to the array data, use the following method:

```
<type>*      getArray() const;
```

Vector Classes

Cosmo 3D provides a wealth of vector math classes. Vectors of different dimensions allow for data categorization according to need, for example, a color value could include four component values. In this case, a four component vector would be used: *csVec4f*.

The following sections describe the vector classes and their transformation class.

Vector Math

Vectors are used in a variety of ways in Cosmo 3D. Commonly, they are used to define orientation, rotation, and transformations. Cosmo 3D provides the following multi-dimensional vectors.

- csVec2f— represents a two-element floating point vector.
- csVec3f— represents a three-element floating point vector.
- csVec3s— represents a three-element short-integer vector.
- csVec4f— represents a four-element floating point vector, often used as a homogenous space coordinate.
- csVec4ub— represents a four-element unsigned byte vector, most often used as a color value. The elements range from 0 to 255, inclusive.

Vector Methods

The methods in the **csVec2f**, **csVec3f**, **csVec3s**, **csVec4f**, classes are similar; the differences between them stem only from additional argument members that account for the different dimensions of the classes. The following discussion describes the **csVec4f** class, but can easily be extended to the other classes as well.

The classes use the following overridden **get()** and **set()** methods to return and define, respectively, the vectors.

```
void        set(float a, float b, float c, float d);
void        get(float *a, float *b, float *c, float *d)  const;
void        set(int i, float f);
float       get(int i) const;
void        set(const csVec4f &v);
void        get(csVec4f &v) const;
```

For the **csVec2f** and **csVec3f** classes, only two or three arguments, respectively, are passed to the set and get methods.

The classes contain the following methods:

```
csBool      equal(const csVec4f&  v) const;
csBool      almostEqual(const csVec4f& v, float tol) const;
void        negate(const csVec4f& v);
float       dot(const csVec4f&  v) const;
void        add(const csVec4f& v1, const csVec4f& v2);
void        sub(const csVec4f& v1, const csVec4f& v2);
void        scale(float s, const csVec4f& v);
void        addScaled(const csVec4f& v1, float s, const csVec4f& v2);
void        combine(float a, const csVec4f& v1, float b, const,
                csVec4f& v2);
float       sqrDistance(const csVec4f& v) const;
float       normalize();
float       length() const;
float       distance(const csVec4f& v) const;
void        xformPt(const csVec4f& v, const csMatrix4f& m);
void        xformVec(const csVec4f& v, const csMatrix4f& m);
```

The methods have the following functionality:

- almostEqual()—returns TRUE if the values are within *tol* of each other.
- negate()—negates the vector, which, in effect, reverses its direction.
- scale()—enlarges or reduces a vector by the multiplier passed in.
- addScaled()—adds to a vector the scaled vector passed in.
- combine()—performs the vector addition of two vectors.
- sqrDistance()—provides the distance squared.
- normalize()—makes the vector orthogonal to its original direction.
- distance()—determines the distance between two vectors.
- xformPt()—transforms the point by the matrix passed in.
- xformVec()—transforms the vector by the matrix passed in.

The classes also provide the following write and operator methods:

```
void        write(csOutput *out);

csVec4f&    operator=(const csVec4f &v);
csVec4f&    operator=(float v);
float&      operator[](int i);
float       operator[](int i) const;
csBool      operator==(const csVec4f &v) const;
csBool      operator!=(const csVec4f &v) const;
```

The **write()** method prints the vector to the device or file defined by *out*.

The **operator()** method defines an operation that can be performed on two vectors. The first two operator methods are assignment operators, the second two are access operators, and the last two are equality operators. For example,

```
csVec2f *myVec = new csVec2f();
csVec2f *yourVec = new csVec2f();
...

if (myVec == yourVec) {...}
```

csVec3s

The methods in **csVec3s** are the same as those in the **csVec3f** class.

csVec4ub Methods

The **csVec4ub** class contains a subset of the above methods, including the **set()**, **get()**, **equal()**, **write()**, and **operator()** methods for two-, three-, and four-dimensional ubyte quantities. For example, the four-dimensional definition is

```
csVec4ub(ubyte a, ubyte b, ubyte c, ubyte d);
```

Transforming csVec3f Vectors

csVec3f vectors are commonly used to specify the placement and orientation of objects in world space. There are two ways to transform a **csVec3f** when passing a **csMatrix4f** into **xform()**:

- A **csVec3f::xformPt(const csVec3f& v, const csMatrix4f& m)** transform vector sets a csVec3f vector, *v*, to be the first three components of (v,1) * *m* where *v* is treated as a row vector.
- **A csVec3f::xformVec(const csVec3f& v, const csMatrix4f& m)** transform point sets this vector to be *v*, thought of as a row vector, times the 3X3 submatrix formed by the first 3 rows and first three columns of *m*. This is most useful for non-projective transformations.

Bounding Volumes

Bounding volumes provide an efficient means of determining which shapes are in and out of the view frustum. The bounding volume for all csNodes is a sphere, as shown in Figure A-1.

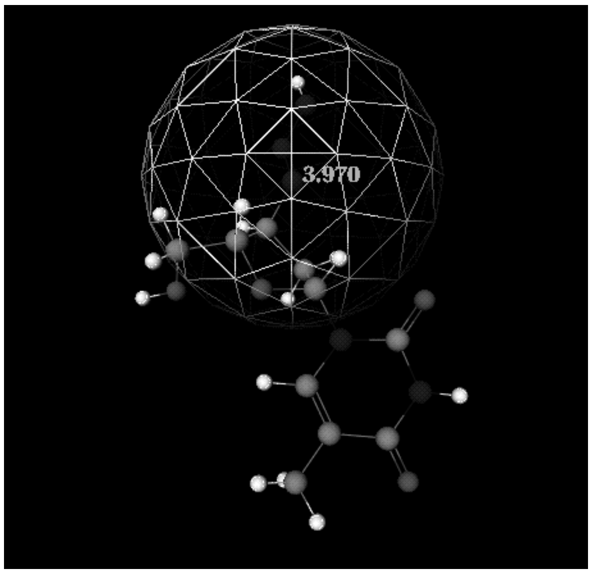

Figure A-1 Bounding Sphere

Cosmo 3D provides two bounding shapes as well as an abstract class from which you can derive your own bounding shapes.

- **csBound**— is the abstract base class from which bounding objects are derived.
- **csBoxBound**—prescribes the minimum-sized box that can enclose an object.
- **csSphereBound**—prescribes the minimum-sized sphere that can enclose an object.

csBound provides methods to compute a bounding object around a group of points, boxes, or spheres, and to extend an existing bounding object by any of these. In addition, there are methods to determine if a point, bounding box, or bounding sphere is contained entirely within a bounding object. Methods are also provided to transform bounding objects using a matrix, test it for emptiness, or test it for intersection with a line segment.

csNodes use **csSphereBound** for efficient bounding updates and checking.

csGeoSets contain a **csBoxBound** because it provides a tighter bound around the vertices.

Field Classes

Together, fields and methods comprise a scene graph node. A field is a class with **set()** and **get()** methods that set and return the values of the field. Fields can also have other methods that perform other operations related to setting or returning the field value.

Cosmo 3D contains the following Field classes:

- **csField**— represents a simple data type, such as float, **csVec3f**, and arrays of simple types.
- **csFieldInfo**—maintains information about the fields of a class including string name, integer id, default value, and a pointer to a member in **csContainer**.
- **csMField**— is an abstract class containing some of the functionality common to arrays.
- **csAtomField**<Type>—is a single-valued atomic field type.
- **csArrayField**<Type>—is a single-valued array field type.

You can substitute for <*Type*> any character type, such as Int, Short, or Field. The following sections describe each of these field classes.

csField

csField is the abstract class from which all of the other field classes are derived. Because it does not have a constructor, you cannot use it directly. See "The csField Class" on page 5 for additional information on **csField**.

csFieldInfo

csFieldInfo maintains information about the fields of a class, including the string name, integer id, default value, and storage offset from class instance, all of which you set in the constructor:

```
csFieldInfo(const char *name, short id, char csContainer::*offset);
```

The class methods provide means of obtaining the argument values.

```
const char*         getName();
short               getId();
char csContainer::* getOffset();
```

csFieldInfo also provides methods that allow you to create an enumeration by pairing an integer with a string, and to create an alias for the field.

```
void                addEnum(int e, const char *str);
void                addAlias(const char *alias);
```

csMField

csMField is an abstract class containing some of the functionality common to arrays, including a container object, an index, and a storage value, as defined by the constructor:

```
csMField(csContainer *p, short i);
```

The set and get methods in the class define array-type functionality, including returning an offset value in the array where Cosmo 3D starts reading the array. The stride function allows you to step through the values in the array at increments of two or more values at a time, for example, if the stride is two, Cosmo 3D uses every other value in the array.

The count and size methods set and return the number of elements in the object, such as the number of values in an array, and the total number of elements allocated for the object.

The **editDone()** method allows you to signal when you have finished manipulating the **csMField** object so that, for example, you can allow its values to take effect.

```
short       getOffset() const;
short       getStride() const;

void        setCount(int n);
int         getCount();

void        setSize(int n);
int         getSize();
void        editDone();
```

csAtomField

csAtomField is a single-valued composite field type. A single value field can be something as simple as an integer or a **csVec4f**.

```
csAtomField<Val>(csContainer *p, short i, Val *stor);
```

csArrayField

csArrayField is a multiple-valued array field type. It is commonly used as the type for array class fields.

```
csArrayField<ValType, ValRef>(csContainer *p, short i,
                              csGenArray<ValType, ValRef> *stor);
```

The class's **set()** and **get()** methods provide the means of setting and returning these values.

```
void        set(int i, ValRef t);
void        get(int i, ValType& t);
ValRef      get(int i);
void        set(const csGenArray<ValType, ValRef> &l);

void        setRange(int i, int count, const ValType vals[]);
void        getRange(int i, int count, ValType vals[]);
void        fillRange(int i, int count, ValRef val);

ValRef      operator[](int i);
```

You can fill an array by setting groups of values using the following methods:

```
void        setRange(int i, int count, csInt vals[]);
void        getRange(int i, int count, csInt vals[]);
void        fillRange(int i, int count, csInt vals[]);
```

- *i* is the position in the array where you want to begin setting values.
- *count* is the number of array elements you want to set.
- *vals[]* is an array of values that you want to enter into the array.

The **fillRange()** method sets all of the values in an array to the value passed in the argument, *vals[]*

The operator, [], assigns values.

```
csInt operator[](int i) const;
```

Other Math Classes

Cosmo 3D includes the following classes used for mathematical calculations.

- csSeg— encapsulates a line segment.
- csPlane—encapsulates a plane.
- csFrustum— encapsulates a viewing frustum.

The following sections explain these classes.

csSeg

csSeg encapsulates a line segment in 3-space as an origin, a normalized direction vector, and a length.

The methods in the class allow you to

- Construct a line from a pair of points.
- Create a vector given a point and polar coordinates.
- Clip a segment out of a line.

csPlane

csPlane represents a half space with a normal and an offset, which together form the parameters of the traditional Ax + By + Cz = D plane equation.

The methods in the class allow you to

- Determine the closest point on the plane to a point in space.
- Determine whether or not it intersects with a **csSeg**.
- Construct a plane from three points.
- Construct a plane from a point and the plane normal at a point.
- Transform the orientation of the plane.

csFrustum

csFrustum is a truncated, possibly asymmetric pyramid for the purposes of testing objects against view volumes.

The methods in the class allow you to

- Determine if a point or shape is in the frustum.
- Copy the contents of the frustum.
- Set and return the near and far clipping planes of the frustum.
- Create an orthogonal frustum.
- Transform the frustum.
- Return the aspect of the frustum and the position of the camera.

Appendix B

Cosmo 3D Sample Application

This appendix discusses a simple Cosmo 3D application, called *cube.cxx*. The files are in the following locations:

	Source	Executable
UNIX	/usr/share/Optimizer/src/apps/Cosmo3D directory	/usr/sbin
NT	C:\Program Files\Silicon Graphics\Optimizer\src\apps\cosmo3D	C:\Program Files...\Optimizer\bind

In *cube.cxx*, two cubes, one red the other green, slowly revolve, as shown (statically) in Figure B-1.

This chapter helps you understand the basic structure of a Cosmo3D application, and explores some of the functionality you can implement in your own applications. The remaining chapters in the book describe the classes and concepts presented in this chapter in greater detail.

These are the sections in this chapter:

- "Cube.cxx Explained" on page 195.
- "Understanding the Different Parts of Cube.cxx" on page 202.
- "Scene Graph for Cube.cxx" on page 203.

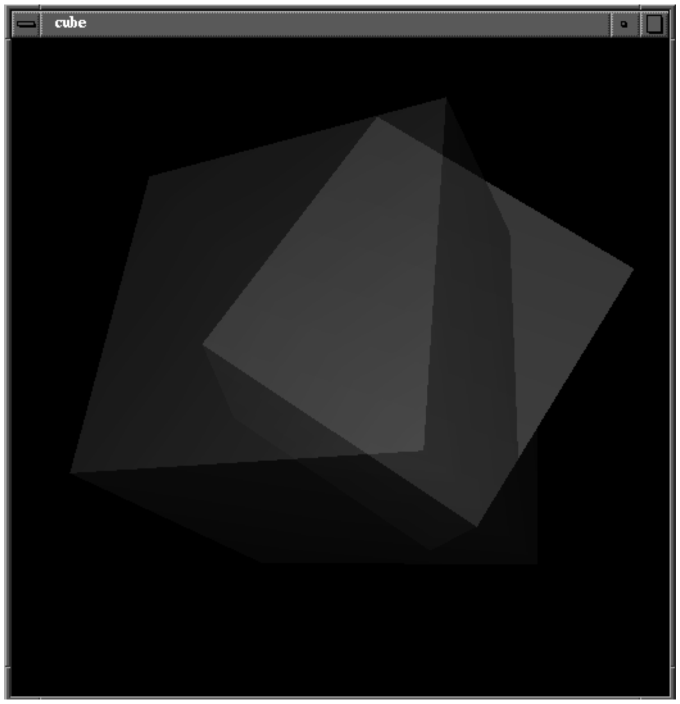

Figure B-1 Cube Application

Cube.cxx Explained

Example B-1 shows the *cube.cxx* application. It also includes embedded comments (not found in the *cube.cxx* file) that explain the functionality of each section of *cube.cxx*. The sections that follow Example B-1 explain the structure and functionality of the code in more generic terms, so that you can understand the principles of programming Cosmo 3D applications.

Example B-1 cube.cxx

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <Cosmo3D/csField.h>
#include <Cosmo3D/csWindow.h>
#include <Cosmo3D/csColorSet.h>
#include <Cosmo3D/csNormalSet.h>
#include <Cosmo3D/csCoordSet.h>
#include <Cosmo3D/csQuadSet.h>
#include <Cosmo3D/csContext.h>
#include <Cosmo3D/csAppearance.h>
#include <Cosmo3D/csPerspCamera.h>
#include <Cosmo3D/csOrthoCamera.h>
#include <Cosmo3D/csTransform.h>
#include <Cosmo3D/csDrawAction.h>
#include <Cosmo3D/csShape.h>
#include <Cosmo3D/csPointLight.h>
#include <Cosmo3D/csMaterial.h>
#include <Cosmo3D/csEnvironment.h>

static csGroup* makeCube();

static csTransform* fgTransform;
static csTransform* bgTransform;
static csAppearance* highlight;
static csGroup* root;
static csDrawAction* da;
static csContext* ctx;

static int doFlash = 0;
static int doBorders = 1;

int frame(void*);
```

```
//Start the application here

int
main(int argc, char *argv[])
{
    int doOrthoCam = 0, doFullScreen = 0;

    if(argc > 1)
    {
        argv++;
    argc--;
        while(argc > 0)
    {
        if(argv[0][0] == '-')
            switch(argv[0][1])
        {
            case 'f':
                doFlash = 1;
            break;
            case 'F':
                doFullScreen = 1;
            doBorders = 0;
            break;
            case 'o':
                doOrthoCam = 1;
            break;
        }

        argv++;
        argc--;
    }
    }
    // Initialize Cosmo3D
    csObject::initClasses();

    // define scene
    root = makeCube();
    // define render window
    if (doFullScreen) {
        csWindow::initPosition(0, 0);
        csWindow::initSize(csWindow::get(csWindow::SCREEN_WIDTH),
                       csWindow::get(csWindow::SCREEN_HEIGHT));
    }
    new csWindow("cube");

    // define rendering context
    csContext *ctx = csWindow::getContext();
    ctx->setDepthEnable(TRUE);
    ctx->setDepthFunc(csContext::LEQUAL_DFUNC);
    ctx->setCullFace(csContext::BACK_CULL);

    //Set up the camera

    csCamera *cam;
    if (doOrthoCam)
    {
    csOrthoCamera *orthoCam = new csOrthoCamera;
        orthoCam->setWidth(4.0f);
        orthoCam->setHeight(4.0f);
        cam = orthoCam;
    }
    else
    {
    csPerspCamera *perspCam = new csPerspCamera;
        cam = perspCam;
    }

    da = new csDrawAction;
    da->setCamera(cam);

    cam->draw(da);

    csWindow::setFrameFunc(&frame, da);
    csWindow::mainLoop();

    return 0;
}
```

```
int
frame(void*)
{
    static int frame = 0;

    const csEventArray &elist = csWindow::getCurrent()->getEvents();
    for (int i=0; i<elist.getCount(); i++)
    {
    if (elist[i]->id == csEvent::KEY_PRESS) {
        switch (elist[i]->key) {
        case 'b':
            doBorders = !doBorders;
        csWindow::setWindowTitle(doBorders? "cube" : NULL);
            break;
        case 's':
        csWindow::reshapeWindow(300, 300);
            break;
        case 'S':
        csWindow::reshapeWindow(600, 600);
            break;
        case 27: // ESC
            exit(1);
        }
    }
    }

    if(frame % 30 == 0 && doFlash)
    ctx->pushOverrideAppearance(highlight);
    else if(frame % 30 == 15 && doFlash)
    ctx->popOverrideAppearance();

    // clear the window
    ctx->clear(csContext::COLOR_CLEAR | csContext::DEPTH_CLEAR);

    // update cube rotations
    bgTransform->setRotation(1.0f, 1.0f, 0.0f, CS_DEG2RAD(frame));
    fgTransform->setRotation(0.5f, 0.1f, 1.0f, CS_DEG2RAD(frame));

    // draw the scene
    da->apply(root);
    // swap buffers
    csWindow::swapBuffers();
    frame++;

    return csWindow::CONTINUE;
}

//Create a cube; the cube will be rendered twice.

static csGroup*
makeCube()
{
    // int i;
    static float cubeCoords[24][3] =
    {
    {-1.0f, -1.0f,  1.0f}, { 1.0f, -1.0f,  1.0f},   // +Z
    { 1.0f,  1.0f,  1.0f}, {-1.0f,  1.0f,  1.0f},   // +Z
    {-1.0f, -1.0f, -1.0f}, {-1.0f,  1.0f, -1.0f},   // -Z
    { 1.0f,  1.0f, -1.0f}, { 1.0f, -1.0f, -1.0f},   // -Z
    { 1.0f, -1.0f,  1.0f}, { 1.0f, -1.0f, -1.0f},   // +X
    { 1.0f,  1.0f, -1.0f}, { 1.0f,  1.0f,  1.0f},   // +X
    {-1.0f, -1.0f,  1.0f}, {-1.0f,  1.0f,  1.0f},   // -X
    {-1.0f,  1.0f, -1.0f}, {-1.0f, -1.0f, -1.0f},   // -X
    {-1.0f,  1.0f,  1.0f}, { 1.0f,  1.0f,  1.0f},   // +Y
    { 1.0f,  1.0f, -1.0f}, {-1.0f,  1.0f, -1.0f},   // +Y
    {-1.0f, -1.0f,  1.0f}, {-1.0f, -1.0f, -1.0f},   // -Y
    { 1.0f, -1.0f, -1.0f}, { 1.0f, -1.0f,  1.0f}    // -Y
    };
    static int numCubeCoords =
sizeof(cubeCoords)/sizeof(cubeCoords[0]);

    static float cubeNorms[6][3] =
    {
    { 0.0f,  0.0f,  1.0f},  // +Z
    { 0.0f,  0.0f, -1.0f},  // -Z
    { 1.0f,  0.0f,  0.0f},  // +X
    {-1.0f,  0.0f,  0.0f},  // -X
    { 0.0f,  1.0f,  0.0f},  // +Y
    { 0.0f, -1.0f,  0.0f}   // -Y
    };
    static int numCubeNorms = sizeof(cubeNorms)/sizeof(cubeNorms[0]);

// Specify the data attributes

    // specify cube as 6 quads
    csQuadSet *gset = new csQuadSet;

    // cube vertices
    csCoordSet3f *cset = new csCoordSet3f(numCubeCoords);
    cset->point()->edit();
#if 0
    for (i=0; i<numCubeCoords; i++)
        cset->point()->set(i,
        csVec3f(cubeCoords[i][0], cubeCoords[i][1], cubeCoords[i][2]));
#else
    cset->point()->setRange(0, numCubeCoords, (csVec3f *)cubeCoords);
#endif
    cset->point()->editDone();
    gset->setCoordSet(cset);

    // cube normals
    csNormalSet3f *nset = new csNormalSet3f(numCubeNorms);
    nset->vector()->edit();
#if 0
    for (i=0; i<numCubeNorms; i++)
        nset->vector()->set(i,
        csVec3f(cubeNorms[i][0], cubeNorms[i][1], cubeNorms[i][2]));
#else
    nset->vector()->setRange(0, numCubeNorms, (csVec3f *)cubeNorms);
#endif
    nset->vector()->editDone();
    gset->setNormalSet(nset);

    gset->setPrimCount(6);
    gset->setNormalBind(csGeoSet::PER_PRIM_NORMAL);

//Specify the appearance and material attributes.

    // highlight, yellow.
    csMaterial *hlMaterial = new csMaterial;
    hlMaterial->setSpecularColor(0.0f, 0.0f, 0.0f);
    hlMaterial->setDiffuseColor(1.0f, 1.0f, 0.0f);
    hlMaterial->setShininess(.0078125 *16.0f);
    hlMaterial->setTransparency(0.0f);
```

```
highlight = new csAppearance;
highlight->setMaterial(hlMaterial);
highlight->setLightEnable(1);
// red cube
csMaterial *redMaterial = new csMaterial;
redMaterial->setSpecularColor(1.0f, 1.0f, 1.0f);
redMaterial->setDiffuseColor(0.8f, 0.1f, 0.1f);
redMaterial->setShininess(.0078125 *16.0f);
redMaterial->setTransparency(0.5f);

csAppearance *redAppearance = new csAppearance;
redAppearance->setMaterial(redMaterial);
redAppearance->setLightEnable(1);
redAppearance->setTranspMode(csContext::BLEND_TRANSP);
redAppearance->setTranspEnable(1);

csShape *redShape = new csShape;
redShape->setAppearance(redAppearance);
redShape->setGeometry(0, gset);

bgTransform = new csTransform;
bgTransform->setTranslation(0.0f, 0.0f, -5.0f);
bgTransform->addChild(redShape);

// green cube
csMaterial *greenMaterial = new csMaterial;
greenMaterial->setDiffuseColor(0.1f, 0.8f, 0.1f);
greenMaterial->setShininess(.0078125 *16.0f);
greenMaterial->setTransparency(0.5f);

csAppearance *greenAppearance = new csAppearance;
greenAppearance->setMaterial(greenMaterial);
greenAppearance->setLightEnable(1);
greenAppearance->setTranspMode(csContext::BLEND_TRANSP);
greenAppearance->setTranspEnable(1);

csShape *greenShape = new csShape;
greenShape->setAppearance(greenAppearance);
greenShape->setGeometry(0, gset);

fgTransform = new csTransform;
fgTransform->setTranslation(0.5f, 0.1f, -6.0f);
fgTransform->addChild(greenShape);
```

```
    // environment
    csPointLight *lt = new csPointLight;
    csEnvironment *environment = new csEnvironment;
    environment->light()->append(lt);
    environment->addChild(fgTransform);
    environment->addChild(bgTransform);

    return environment;
}
```

Understanding the Different Parts of Cube.cxx

The embedded comments in Example B-1 call out the different functional parts of *cube.cxx*, which include:

- Include statements and global method declarations.
- Create a window in which the application runs and with which a user can interact with the application. For more information, see Chapter 12, “User Interface Mechanisms.”
- Instantiate and setup a camera. For more information, see Chapter 9, “Viewing the Scene.”
- Create a scene graph. For more information, see Chapter 4, “Scene Graph Nodes.”
- Draw and rotate the cubes represented by the data in the scene graph. For more information, see Chapter 6, “Placing Shapes in a Scene.”

Scene Graph for Cube.cxx

Scene graphs provide the structure for Cosmo 3D applications. Cosmo 3D applications use scene graphs to specify the objects rendered. Figure B-2 shows the scene graph used for *cube.cxx*.

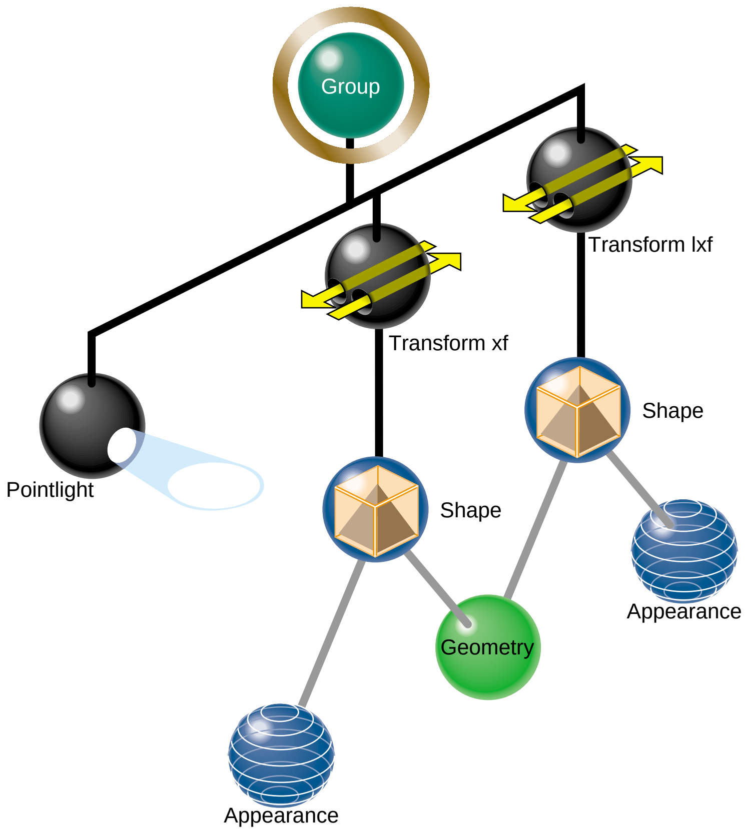

Figure B-2 Cube Scene Graph

csShape nodes define a shape; they associate a **csAppearance**, which describes the look of a shape (such as its color), with a **csGeometry**, which defines the dimensions of the geometry (such as whether the geometry is a cube or sphere).

In *cube.cxx*, **csGeometry** defines a cube and the **csAppearance** nodes specify the green and red colors of the cubes.

Note: Neither **csGeometry** nor **csAppearance** are nodes; they are classes associated by a **csShape** node.

Relating Local Space to World Space

Once you define the orientation of a shape, you use **csTransform** nodes to place and orient the shape in a different coordinate system. *World space* is the coordinate system of the root node. If all the shapes in a scene graph are transformed into world space, a **csCamera** object attached to the root node can view all the shapes in the scene graph together in one coordinate system.

Objects in world space are rendered when a draw action is applied to the root node of the scene graph. Objects in local space are rendered when a draw action is applied to a subsection of the scene graph. The same object rendered in these two spaces may appear different, for example, a shape in world space may appear smaller than in local space because it is farther from the viewer; it might also be rotated and positioned differently.

Generally, there are many transformation nodes in a scene graph and a shape is often transformed more than once, as shown in Figure B-3.

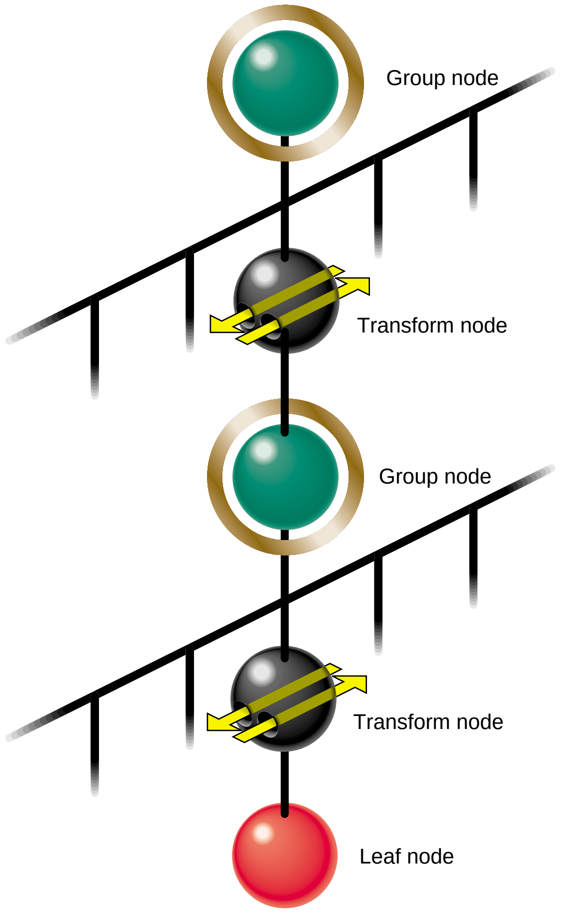

Figure B-3 Two Transformations Into World Space

In Figure B-3, after the leaf node is transformed twice, it is placed in world space.

In *cube.cxx*, two transformation nodes transform the shapes into world space.

Creating the User Interface

csWindow encapsulates the user interface: it includes the methods you use to construct a window in which a Cosmo 3D application runs. **csWindow** manages a **csContext** object to control the graphics context as well as a **csEvent** object that handles user actions, like mouse events.

csWindow is replete with default values that satisfy most application needs. *cube.cxx*, for example, uses all of the default values except for the title of the window, which is specified with the **InitWindow()** method.

Cosmo 3D also allows you to construct your own window using X window code. This option gives you complete control over the look and functionality of the window.

For a complete description of **csWindow** and using X window code, see Chapter 12, "User Interface Mechanisms."

Rendering World Space

To render a scene graph, the **csDrawAction::apply()** method is applied to the root node of the scene graph, as follows:

```
da->apply(root);
```

where *da* is a **csDrawAction** object.

Summary

The following procedure summarizes the steps you take to create and render a very simple scene graph.

1. Create **csAppearance** and **csGeometry** containers to define the appearance and the geometry of an object. For more information on setting **csAppearance** values, see Chapter 3, "Specifying the Appearance of Geometries." For more information on setting **csGeometry** values, see Chapter 2, "Creating Geometries."
2. Relate the **csAppearance** and **csGeometry** nodes in a **csShape** node. For more information on setting **csShape** values, see Chapter 2, "Creating Geometries."
3. Add the **csShape** nodes as children of the **csTransform** nodes. The **csTransform** node orients and positions the **csShape** objects in world space. For more information on setting **csTransform** values, see Chapter 6, "Placing Shapes in a Scene."

   Note: A **csShape** node by itself can be a complete scene graph. Typically, however, scene graphs have many **csShape** nodes, most of which are connected to other parts of the scene graph with a **csTransform** nodes.
4. Add the **csTransform** nodes to the scene graph. For more information about adding nodes to scene graphs, see Chapter 6, "Placing Shapes in a Scene."
5. Create a window, **csWindow**, in which to view and interact with the application.
6. Set the current graphical context, **csContext**.
7. Draw all of the shapes in world space by applying a **csDrawAction** to the root of the scene graph. For more information about draw actions, see Chapter 7, "Traversing the Scene Graph."

Appendix C

Cosmo 3D Class Hierarchy

The following list shows the class hierarcy in Cosmo 3D.

```
csActionFunTable
csArray
    csByteArray
    csFloatArray
    csIntArray
    csMatrix4fArray
    csPtrArray
        csEventArray
        csFieldArray
        csFieldInfoArray
        csRefArray
    csRotationArray
    csShortArray
    csStringArray
    csVec2fArray
    csVec3fArray
    csVec3sArray
    csVec4fArray
csBitMask
csBound
    csBoxBound
    csSphereBound
csCloneTable
csDict
csDictEntry
```

```
csField
    csMField
        csMFByte
        csMFFloat
        csMFInt
        csMFMatrix4f
        csMFRef
        csMFRotation
        csMFString
        csMFVec2f
        csMFVec3f
        csMFVec3s
        csMFVec4f
    csSFBitMask
    csSFBoxBound
    csSFDouble
    csSFEnum
    csSFFloat
    csSFInt
    csSFMatrix4f
    csSFName
    csSFRef
    csSFRotation
    csSFShort
    csSFSphereBound
    csSFString
    csSFTime
    csSFUByte
    csSFUInt
    csSFVec2f
    csSFVec3f
    csSFVec3s
    csSFVec4f
    csSFVec4ub
```

```
csFieldInfo
    csMFByteInfo
    csMFFloatInfo
    csMFIntInfo
    csMFMatrix4fInfo
    csMFRefInfo
    csMFRotationInfo
    csMFStringInfo
    csMFVec2fInfo
    csMFVec3fInfo
    csMFVec3sInfo
    csMFVec4fInfo
    csSFBitMaskInfo
    csSFBoxBoundInfo
    csSFDoubleInfo
    csSFEnumInfo
    csSFFloatInfo
    csSFIntInfo
    csSFMatrix4fInfo
    csSFNameInfo
    csSFRefInfo
    csSFRotationInfo
    csSFShortInfo
    csSFSphereBoundInfo
    csSFStringInfo
    csSFTimeInfo
    csSFUByteInfo
    csSFUIntInfo
    csSFVec2fInfo
    csSFVec3fInfo
    csSFVec3sInfo
    csSFVec4fInfo
    csSFVec4ubInfo
csFrustum
csGlobal
csMatStack4f
csMatrix4f
csName
csNameEntry
```

```
csObject
    csAction
        csCompileAction
        csTransformAction
            csIsectAction
            csSoundAction
            csVFCullAction
                csDrawAction
    csContainer
        csAppearance
        csAudio
        csAudioClip
        csAudioSamples
        csCamera
            csFrustumCamera
            csOrthoCamera
            csPerspCamera
        csColorSet
            csColorSet3f
            csColorSet4f
        csCoordSet
            csCoordSet3f
        csGeometry
            csBox
            csCone
            csCylinder
            csGeoSet
                csLineSet
                csLineStripSet
                    csIndexedLineSet
                csPointSet
                csPolySet
                    csIndexedFaceSet
                csQuadSet
                csTriFanSet
                csTriSet
                csTriStripSet
            csScreenAlignedText
            csSphere
        csImage
        csIndexSet
        csMaterial
        csMicrophone
```

```
csNode
    csEngine
        csInterpolator
            csColorInterpolator
            csCoordinateInterpolator
            csNormalInterpolator
            csOrientationInterpolator
            csPositionInterpolator
            csScalarInterpolator
        csMorphEng
            csMorphEng3f
            csMorphEng4f
        csSelectorEng
            csSelectorEng3f
            csSelectorEng4f
        csSpline
        csTransformEng
            csTransformEng3f
    csFog
    csGroup
        csBillboard
        csCollision
        csEnvironment
        csSwitch
        csSwitch     csInline
        csSwitch     csLOD
        csTransform
    csLight
        csDirectionalLight
        csPointLight
            csSpotLight
    csPickSensor
        csPlaneSensor
        csSphereSensor
        csTouchSensor
    csShape
    csSound
    csTimeSensor
csNormalSet
    csNormalSet3f
    csNormalSet3s
csSoundPlayer
```

```
        csTexCoordSet
            csTexCoordSet2f
        csTexGen
        csTexture
            csImageTexture
    csContext
    csData
    csDispatch
    csEvent
    csHit
    csOverrideGeoProp
    csWindow
csOgl
csOutput
csPrivate
csPtrArray2D
csRotation
csSeg
csString
csStringDict
csStringDictEntry
csThread
csTime
csType
csVec2f
csVec3f
csVec4f
csVec4ub
csdFile_csb
vlDB
vlDebugError
vlGeom
vlInput
vlReadError
```

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3445-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389