Cosmo 3D™ Programmer's Guide

Document Number 007-3445-001

CONTRIBUTORS

Written by George Eckel
Illustrated by Dany Galgani
Engineering contributions by John Rohlf, Brad Grantham, Chris Tanner, Rich Silba, Michael Jones.

Cosmo 3D™ Programmer's Guide
Document Number 007-3445-001

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 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.

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 *cosmo1.0/cosmo/test/C++* directory.

Brief descriptions of the chapters in this guide follow:

- Chapter 1, "Introduction to Cosmo 3D," provides an overview of Cosmo 3D.
- Chapter 2, "Your First Cosmo 3D Application," introduces you to a simple Cosmo 3D application that is included in the *cosmo1.0/cosmo/test/C++* directory, called *cube.cxx*.
- Chapter 3, "Creating Geometries," discusses large, ready-made geometries, such as **csSphere** and **csCube** objects, how to create your own **csGeoSet**-derived classes, and how to use the **csGeoSet**-derived classes provided by Cosmo 3D.
- Chapter 4, "Specifying the Appearance of Geometries," describes the appearance fields in **csContext** and **csAppearance**.
- Chapter 5, "Scene Graph Nodes," describes nodes and node types.
- Chapter 6, "Building a Scene Graph," describes how to build and edit a scene graph.

- Chapter 7, “Placing Shapes in a Scene,” describes how to place shapes in scenes.
- Chapter 8, “Traversing the Scene Graph,” describes how an action traverses a scene graph and a description of the actions available in Cosmo 3D.
- Chapter 9, “Lighting,” describes how to use lights, change the shadow modeling, and change the screen to one color.
- Chapter 10, “Viewing the Scene,” describes how to set up the viewport and how to use cameras to view a scene.
- Chapter 11, “Scene Graph Engines,” describes **csEngine** and the multiple subclasses derived from it
- Chapter 12, “User Interface Mechanisms,” discusses how to implement user interaction using X window code, **csWindow**, and selection mechanisms.
- Chapter 13, “Optimizing Rendering,” describes the Cosmo 3D nodes and programming techniques that can help optimize your application’s performance.
- Chapter 14, “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.

These chapters are followed by an index.

Who Should Read This Guide

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

What You Should Know Before Reading This Book

This guide is written with the assumption that the reader is experienced with

- C++
- Character animation

This book does not review C++ programming techniques; it is assumed that the reader can read and program in C++. This book does not cover character animation directly.

Suggestions for Further Reading

For information about 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*. Mass.:Addison Wesley 1994

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

Introduction to Cosmo 3D

Cosmo 3D is a toolkit that brings 3D graphics programming to desktop applications. The toolkit is designed to provide fast, general 3D rendering across a wide range of platforms, from PCs to workstations. Cosmo 3D defines a C++ interface for high-performance, 3D desktop applications.

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

These are the sections in this chapter:

- "Cosmo 3D Overview" on page 1.
- "Cosmo 3D Contents" on page 2.

Cosmo 3D Overview

Cosmo 3D is a platform-independent graphics toolkit that brings ultra-high performance, real-time, 3D applications to the Internet and the desktop.

Cosmo 3D is independent of its underlying rendering architecture enabling maximum portability across all hardware platforms. Developed concurrently on both mainstream personal computers (PCs), as well as high-end professional workstations, Cosmo 3D provides extensive stability.

Cosmo 3D provides high-performance 3D graphics technology for developers of powerful desktop applications. With advanced features, such as a scene graph architecture, morphing, view culling, level of detail (LOD), 2D texture mapping, and spatialized audio, Cosmo 3D enables you to develop, for example, professional character animations and gaming applications.

Cosmo 3D Contents

The following list of class descriptions provides an overview of the functionality of the Cosmo 3D library.

Fundamentals

- Scene Graph - A directed acyclic graph (DAG) of nodes, which represent a database.
- Node - The abstract base class for objects that may be connected to and acted upon in a scene graph.
- Fields - Classes or simple data types that set and return node values.
- Type - Specifies the fields of a class by listing the fields that an instance of the class contains.

Graphics State

- Context - Manages the graphics state of the rendering engine.
- Appearance - Encapsulates the graphics attributes that define the appearance of a **csGeometry** object.
- Material - Defines the light reflectance characteristics of a surface.
- Texture - Defines an image that may be applied to a surface for increased detail.
- TexGen - Automatically generates texture coordinates from vertex coordinates.

Geometry

- Geometry - The abstract base class for geometric primitives.
- GeoSet - A collection of like primitives.
- PointSet - A collection of equally-sized points.
- LineStripSet - A collection of linestrips (polylines) of equal width.
- TriStripSet - A collection of triangle strips.
- TriFanSet - A collection of triangle fans.
- PolySet - A collection of polygons.

- IndexedFaceSet - A set of polygons.
- IndexedLineSet - A set of polylines.
- Sprite - A rectangle which is rotated to face the viewer.

Grouping Nodes

- Group - A node that may have other nodes as children.
- Switch - A group node that selects none, one, or all of its children depending on its value.
- LOD - A switch node that selects among its children based on an evaluation function, such as the distance between the viewer and a shape.
- Transform - Translates the coordinate system of its children into that of its parent node.
- Environment - A grouping node that defines the scope and effect of Light and Fog.

Leaf Nodes

- Shape - A leaf node that associates a **csGeometry** object with a **csAppearance** object.
- Light - An abstract base class for light sources.
- DirectionalLight - A directional light source whose origin is at infinity.
- PointLight - A point light source that radiates equally in all directions.
- SpotLight - A conical spotlight.
- Fog - Defines the atmospheric attenuation of light.
- Sound - Defines a spatialized sound.

Engines

- Spline - Interpolates an arbitrary, non-uniform spline and outputs a weight array.
- MorphVec - An Engine that produces a weighted sum of attribute sets.
- TransformVec - An Engine that transforms a set of coordinates.

Sound

- AudioClip - Contains state information about how a sound file should be played.
- AudioSamples - Contains actual sound samples in a specified format.

Traversals

- Action - An abstract traversal that maintains state and transformation stacks.
- DrawAction - Renders a scene graph.
- IsectAction - Intersects a set of line segments with a subgraph.
- SoundAction - Plays sound specified in **csSound** nodes.

Cameras and Viewing

- Camera - An abstract base class that defines the viewing parameters used when rendering and picking.
- OrthoCamera - Defines an orthographic projection.
- PerspCamera - Defines a perspective projection whose frustum is symmetric.

Basic Classes

- BitMask - A collection of bits settable individually or as a range.
- Data - A raw, untyped storage similar in spirit to void *.
- Array - Optionally interleaved array classes.
- Vec2f - A two-element floating point vector.
- Vec3f - A three-element floating point vector.
- Vec4f - A four-element floating point vector.
- Matrix4f - A 4x4 floating point matrix.
- Name - A descriptive string.
- Rotation - Axis-amount representation of a rotation.
- Bound - Abstract bounding volume.

- BoxBound - A rectangular solid bounding volume.
- SphereBound - A spherical bounding volume.
- Frustum - Truncated pyramid used for culling.
- Seg - Line segment used for intersection.

Geometry Attributes

- CoordSet - A set of coordinates.
- NormalSet - A set of normals.
- ColorSet - A set of colors.
- TexCoordSet - A set of texture coordinates.
- IndexSet - An optional set of indices for indirection into an attribute set.

Field 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, MFFloat, MFMatrix4f, MFVec2f, MFVec3f, MFVec4f, MFRotation.

Chapter 2

Your First Cosmo 3D Application

This chapter introduces you to a simple Cosmo 3D application that is included in the /trees/*cosmo/cosmo/test/C++* directory, called *cube.cxx*. The executable, *cube*, is also included in the same directory so you can see what the application looks like, as shown in Figure 2-1.

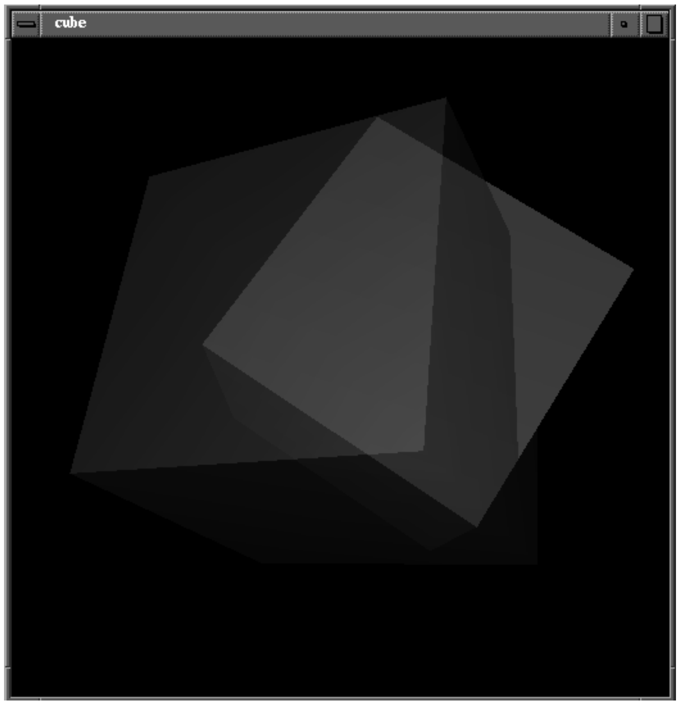

Figure 2-1 Cube Application

In *cube.cxx*, two cubes, one red the other green, slowly revolve.

This chapter gives you the feeling of Cosmo 3D, the basic structure of its applications, and some of the functionality you can look forward to implementing 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 8.
- "Understanding the Different Parts of Cube.cxx" on page 16.
- "Scene Graph" on page 16.

Cube.cxx Explained

Example 2-1 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 2-1 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 2-1 cube.cxx

```
// Most Cosmo 3D classes comprise their own file; to use them, include
// their header files at the top of the application

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>
```

```
#define RED
//#define GREEN
#define FUNKY

static csGroup*         makeCube();

static csTransform*     fgTransform;
static csTransform*     bgTransform;
static csTransform*     xxTransform;
static csAppearance*    highlight;
static csEnvironment*   environment;

// Start the application here

int
main(int argc, char *argv[])
{
    // Parse the arguments; (-o) specifies using an orthocam
    // otherwise use a perspcam

int doOrthoCam = 0;
    int doFlash = 0;

    if(argc > 1)
    {
        argv++;
        argc--;
        while(argc > 0)
        {
            if(argv[0][0] == ‘-’)
                switch(argv[0][1])
                {
                    case ‘f’:
                        doFlash = 1;
                        break;
                    case ‘o’:
                        doOrthoCam = 1;
                        break;
}

            argv++;
            argc--;
        }
    }
    /* define scene */
    csGroup *root = makeCube();

    /* define render window */
    new csWindow(“cube”);

    /* define rendering context */
    csContext *cxt = new csContext;
    cxt->setDepthFunc(csContext::LEQUAL_DFUNC);
    cxt->setCullFace(csContext::BACK_CULL);
    cxt->makeCurrent();

    // csContext::setDepthFunc(csContext::LEQUAL_DFUNC);
    // csContext::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;
    }

    csDrawAction *da = new csDrawAction;
    da->setCamera(cam);

    cam->draw();
    while(1)
    {
        static int frame = 0;

        if(frame % 30 == 0 && doFlash)
            csContext::pushOverrideAppearance(highlight);
        else if(frame % 30 == 15 && doFlash)
            csContext::popOverrideAppearance();
```

```
#if 0
        if (frame == 60)
           environment->addChild(xxTransform);

#endif

        /* clear the window */
        csContext::clear(csContext::COLOR_CLEAR |
csContext::DEPTH_CLEAR);
```

// Render and rotate the cube in real space

```
#ifdef RED
        bgTransform->setRotation(1.0f, 1.0f, 0.0f, CS_DEG2RAD(frame));
#endif
#ifdef GREEN
        fgTransform->setRotation(0.5f, 0.1f, 1.0f, CS_DEG2RAD(frame));
#endif
#ifdef FUNKY
        xxTransform->setRotation(0.2f, 1.0f, 0.3f, CS_DEG2RAD(frame));
#endif

        /* draw the scene */
        da->apply(root);

        /* swap buffers */
        csWindow::swapBuffers();
        frame++;
    }
}
```

// Create a cube; the cube will be rendered twice

```
static csGroup*
makeCube()
{
    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]);

    static float cubeColors[6][3] =
    {
    { 1.0f,  0.0f,  1.0f},
    { 0.0f,  0.0f,  1.0f},
    { 1.0f,  0.0f,  0.0f},
    { 1.0f,  1.0f,  0.0f},
    { 0.0f,  1.0f,  0.0f},
    { 1.0f,  1.0f,  0.0f}
    };
    static int numCubeColors =
sizeof(cubeColors)/sizeof(cubeColors[0]);
```

// Specify the data attributes

```
/*
     * specify cube as 6 quads
     */

    csQuadSet *gset = new csQuadSet;

    /* cube vertices */
    csCoordSet3f *cset = new csCoordSet3f(numCubeCoords);
    csVec3f *coords = cset->point()->edit();
    for (int i=0; i<numCubeCoords; i++)
       cset->point()->set(i,
           csVec3f(cubeCoords[i][0], cubeCoords[i][1],
cubeCoords[i][2]));
    cset->point()->editDone();
    gset->setCoordSet(cset);

    /* cube normals */
    csNormalSet3f *nset = new csNormalSet3f(numCubeNorms);
    csVec3f *norms = nset->vector()->edit();
    for (i=0; i<numCubeNorms; i++)
       nset->vector()->set(i,
           csVec3f(cubeNorms[i][0], cubeNorms[i][1], cubeNorms[i][2]));
    nset->vector()->editDone();
    gset->setNormalSet(nset);

    /* cube colors */
    csColorSet3f *clrset = new csColorSet3f(numCubeColors);
    csVec3f *colors = clrset->color()->edit();
    for (i=0; i<numCubeColors; i++)
       clrset->color()->set(i,
           csVec3f(cubeColors[i][0], cubeColors[i][1],
cubeColors[i][2]));
    clrset->color()->editDone();

    gset->setPrimCount(6);
    gset->setCullFace(csContext::NO_CULL);
    gset->setNormalBind(csGeoSet::PER_PRIM_NORMAL);
    gset->setColorBind(csGeoSet::NO_COLOR);
```

// 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);

#ifdef RED
    /* 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);
#endif

#ifdef GREEN
    /* green cube */
    csMaterial *greenMaterial = new csMaterial;
    greenMaterial->setSpecularColor(1.0f, 1.0f, 1.0f);
    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);
#endif

#ifdef FUNKY
```

```
    /* funky cube with all sorts of fun colors */
    csQuadSet *funky_gset = new csQuadSet;
    funky_gset->setCoordSet(cset);
    funky_gset->setNormalSet(nset);
    funky_gset->setColorSet(clrset);
    funky_gset->setPrimCount(6);
    funky_gset->setNormalBind(csGeoSet::PER_PRIM_NORMAL);
    funky_gset->setColorBind(csGeoSet::PER_PRIM_COLOR);

    csAppearance *funkyAppearance = new csAppearance;
    funkyAppearance->setLightEnable(1);
    funkyAppearance->setTranspMode(csContext::BLEND_TRANSP);
    funkyAppearance->setTranspEnable(1);

    csShape *funkyShape = new csShape;
    funkyShape->setAppearance(funkyAppearance);
    funkyShape->setGeometry(0, funky_gset);

    xxTransform = new csTransform;
    xxTransform->setTranslation(-0.5f, -0.1f, -6.0f);

xxTransform->addChild(funkyShape);
#endif

    /* environment */
    csPointLight *lt = new csPointLight;
    environment = new csEnvironment;
    environment->light()->append(lt);
#ifdef GREEN
    environment->addChild(fgTransform);
#endif
#ifdef FUNKY
    environment->addChild(xxTransform);
#endif
#ifdef RED
    environment->addChild(bgTransform);
#endif
    return environment;
}
```

Understanding the Different Parts of Cube.cxx

The embedded comments in Example 2-1 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.
- Instantiate and setup a camera.
- Create a scene graph.
- Draw and rotate the cubes represented by the data in the scene graph.

Scene Graph

Scene graphs provide the structure for Cosmo 3D applications. Cosmo 3D applications use scene graphs to specify the objects rendered. Figure 2-2 shows the scene graph used for *cube.cxx*.

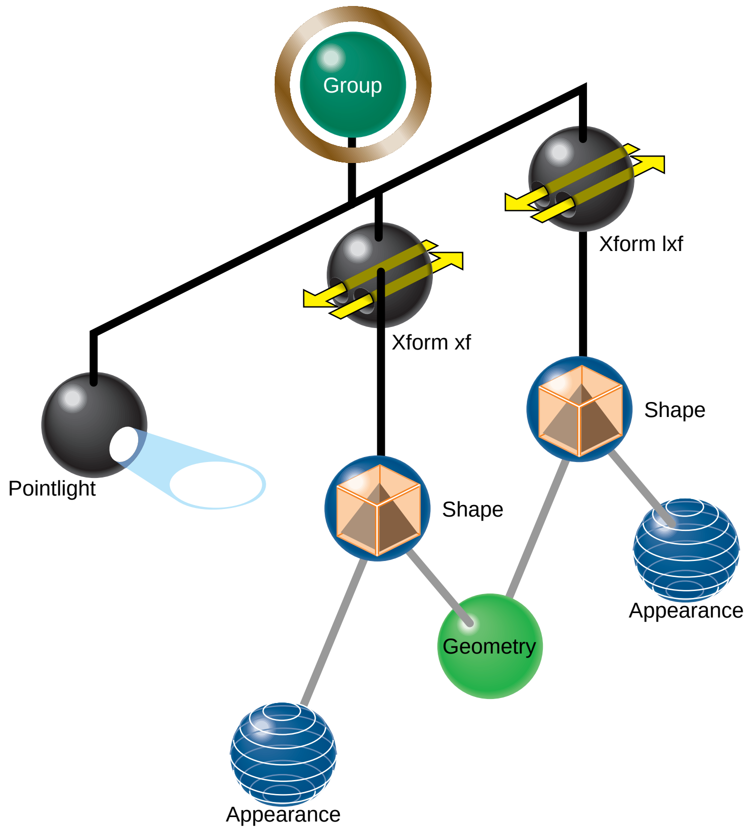

Figure 2-2 Cube Scene Graph

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 **csTransformation** nodes to place and orient the shape in a different coordinate system. *World space* is the coordinate system of the root node. If all of the shapes in a scene graph are transformed into world space, a **csCamera** object attached to the root node can view all of 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.

Generally, there are many transformation nodes in a scene graph and a shape is often transformed more than once, as shown in Figure 2-3.

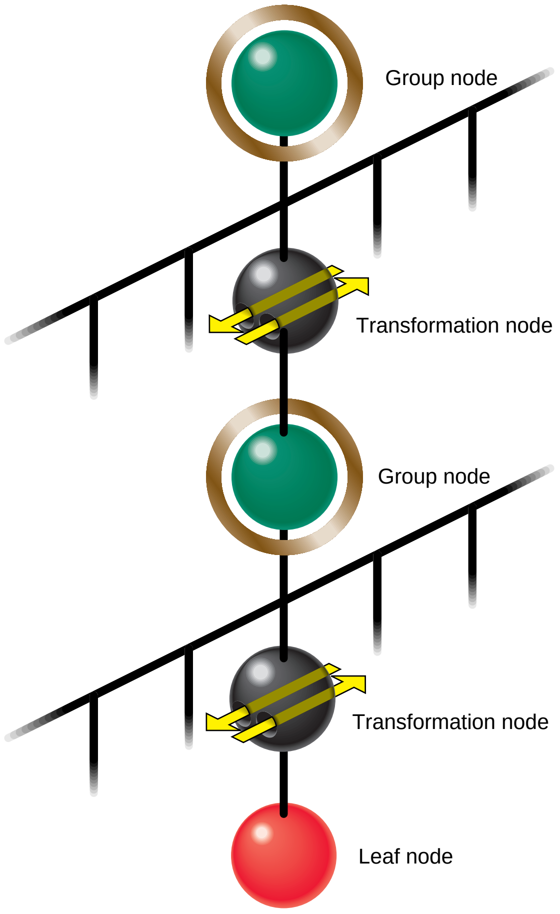

Figure 2-3 Two transformation into World Space

In Figure 2-3, 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 4, "Specifying the Appearance of Geometries." For more information on setting **csGeometry** values, see Chapter 3, "Creating Geometries."
2. Relate the **csAppearance** and **csGeometry** nodes in a **csShape** node. For more information on setting **csShape** values, see Chapter 3, "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 7, “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 7, “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 8, “Traversing the Scene Graph.”

Now that you understand the general organization of the nodes in a scene graph, you need to know how to set the values for the nodes.

Chapter 3

Creating Geometries

csGeometry is a virtual class. All derivations of the class represent one or more shapes, such as a sphere, cube, or 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 **csContext** object contains the default appearance characteristics used for all geometries. You use **csAppearance** objects to customize the default appearance settings for shapes.

In this book,

- A *geometry* is an object of any form; the surface of which is uniform and non-descript, encapsulated in **csGeometry** objects or objects derived from this class.
- An *appearance* contains all the parameters that specify the look of a geometry, encapsulated in a **csAppearance** object.
- A *shape* is a combination of a geometry and an appearance, encapsulated in a **csShape** node.

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.

The first part of this chapter discusses large, ready-made geometries, such as **csSphere** and **csCube** objects. 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:

- "Using Large Geometries" on page 24.
- "csGeoSet Attributes" on page 27.

- “Setting Attributes” on page 29.
- “Cosmo 3D-Derived csGeoSet Objects” on page 36.

Using Large Geometries

Cosmo 3D comes with four ready-made geometries:

- csSphere
- csCube
- csBox
- csCone

Each of these classes have methods that allow you to set and retrieve the values necessary to define the geometry, including, where appropriate,

- the 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, squares, or triangle strips. All of the primitives in a collection are of the same type. You can construct a shape by specifying the coordinates of each of these primitives 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 is captured as attributes of each primitive.

Each **csGeoSet**-derived object contains an array of primitive shapes, each primitive is made of an array of four attributes, and each of the four attributes refers to an array of attribute values, as shown in Figure 3-1.

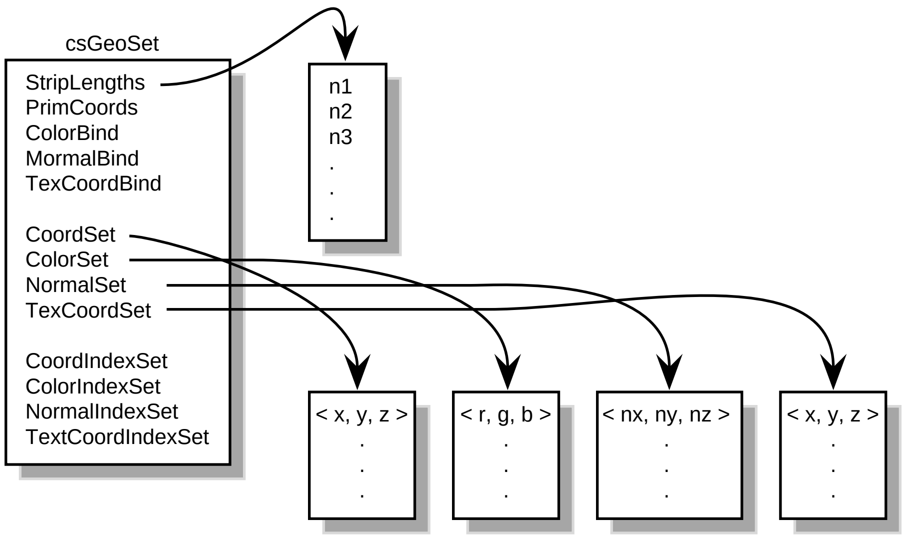

Figure 3-1 Primitives in a csGeoSet

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

csGeoSet Fields

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

```
// General settinge
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.

Setting the Number of Primitives

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

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

To specify or return the number of primitives in a **csGeoSet** object, use the **setPrimCount()** and **getPrimCount()** methods, respectively.

csGeoSet Attributes

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

- **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.
- **csSprite**—A rectangle rotated to face the viewer.

All of the primitives in each of their classes are of equal size. These primitives are constructed from 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; one primitive is defined by twelve attribute values.

Note: Although texture coordinates can be specified using four values, S, T, R, Q, the R value has no current meaning and the Q value is always one.

Attribute Bindings

Not all attributes, however, 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 of these levels 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 3-1 shows the different possible bindings.

Table 3-1 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 primitives 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.

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 3-1.

```
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 3-1, **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 3-2.

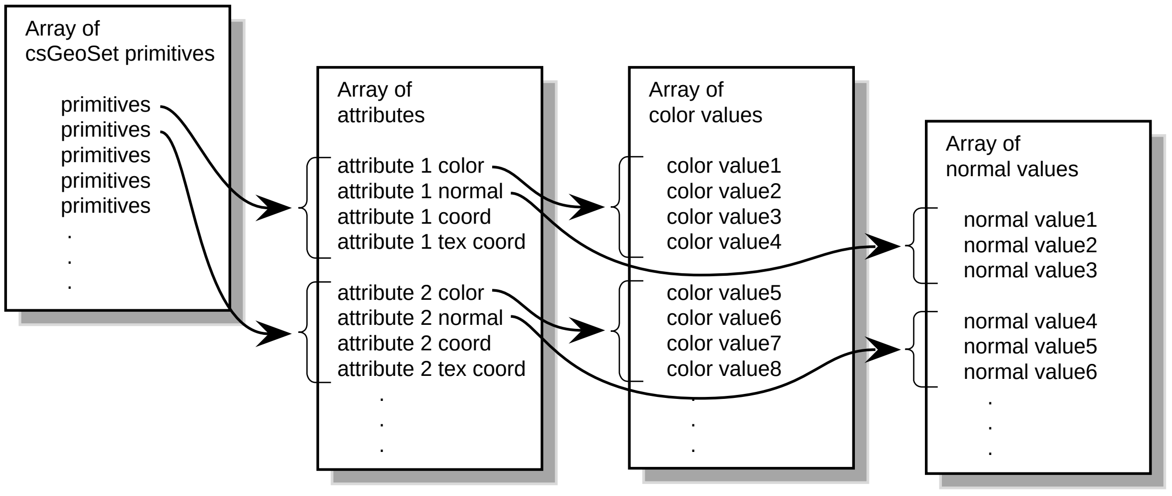

Figure 3-2 Sequential Specification of Attributes Per Primitive

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 3-3.

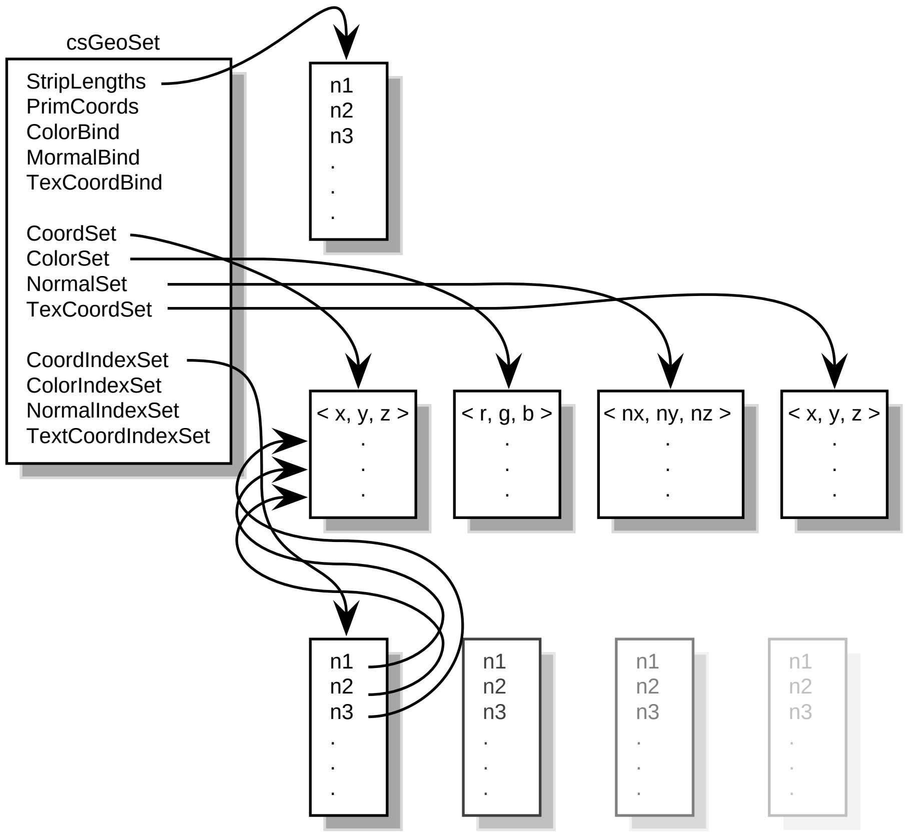

Figure 3-3 Indexed Attributes

When to Index Attributes

The choice of using indexed or sequential attributes applies to all of the primitives in a **csGeoSet**; that is, all of the primitives within one **csGeoSet** must be referenced sequentially or by index; you cannot mix the two.

The governing principle for indexing attributes or not is how many vertices in a geometry are shared. Consider the following two examples in Figure 3-4 where each dot marks a vertex.

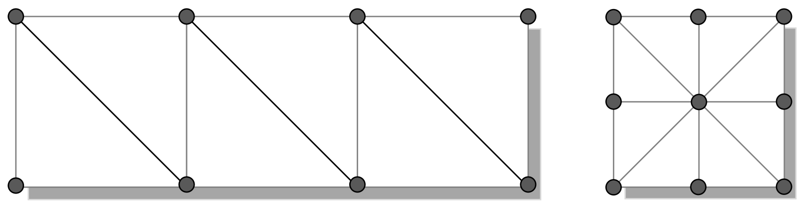

Figure 3-4 Deciding Whether to Index Attributes

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 34 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.

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 constructors 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* specifies the grouping of array primitives when reading the data. If the stride value is one, every color value is read. If the stride value is three, only every third value is read.

The normal implementation is to set the offset to the number into the primitive array of the value you want to access, such as the red value of the first vertex. The stride value is then set to the number of array elements between the red array element and the next array element, as shown in Figure 3-5.

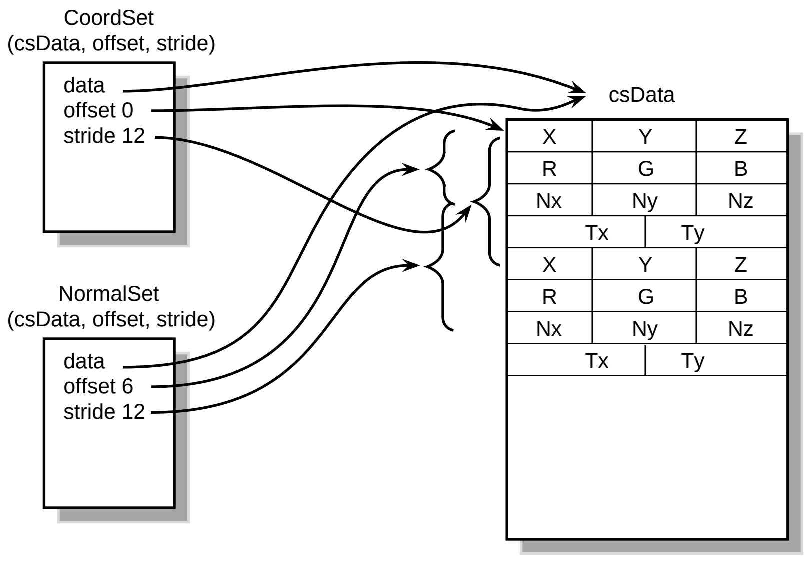

Figure 3-5 Stride and Offset Values

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 of set and get methods use the following form:

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

Setting Attributes Sequentially

To use the attribute values sequentially in their arrays, you set all of the **set...Indices()** methods to NULL. Because the Indices array is NULL, the array of primitives maps directly to the array of attribute values, as shown in Figure 3-2.

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 3-3.

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 3-1 shows how to set attributes and their bindings.

Example 3-1 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->setCoordIndices(iset);
gset->setNormals(nset);
gset->setColors(cset);
// Set the attribute indices
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();
```

Cosmo 3D-Derived csGeoSet Objects

Cosmo 3D provides the following ready-made **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, of equal width.
- **csTriSet**—A collection of triangles.
- **csTriFanStrip**—A collection of triangles that all share one vertex point.
- **csTriStripSet**—A collection of triangle strips.
- **csPolySet**—A collection of convex, coplanar polygons.
- **csSprite**—A rectangle rotated to face the viewer.
- **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 **boundingbox()** methods. The **draw()** method handles a draw; it specifies how a csGeoSet object is drawn.

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 methods:

```
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 member functions allow you to set and return the length of the lines in the collection.

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

Using csIndexedLineSet

A **csIndexedLineSet** object contains an indexed collection of lines of equal length. The member functions 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 methods:

```
void        setStripLengths(csIndexSet* stripLengths);
csIndexSet* getStripLengths();

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

The **setStripLengths()** and **getStripLengths()** methods allow 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 lines of equal length.

Using csTriFanSet

A **csTriFanSet** is a set of triangles all of which share one common vertex, as shown in Figure 3-6.

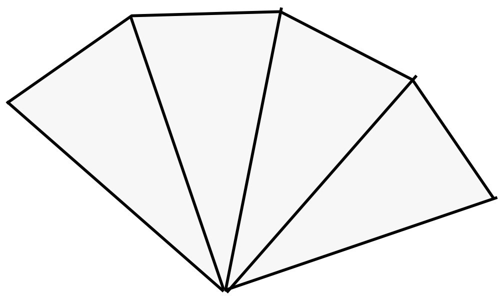

Figure 3-6 TriFanSet

You use the following method to 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 3-7.

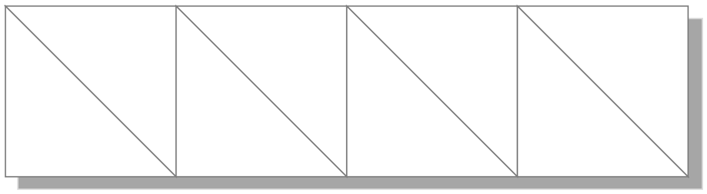

Figure 3-7 Triangle Strip

csTriStripSet contains the following methods:

```
void        setStripLengths(csIndexSet* stripLengths);
csIndexSet* getStripLengths();
```

The **setStripLengths()** and **getStripLengths()** methods allow you to specify and find out, respectively, how long, in pixels, each triangle strip is in a **csTriStripSet** object.

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:

```
void        setPolyLengths(csIndexSet* polyLengths);
csIndexSet* getPolyLengths();
```

The **setPolyLengths()** and **getPolyLengths()** methods allow you to specify and find out, respectively, how long, in pixels, all of the polygon sides are in a **csPolySet** object.

Using csSprite

A **csSprite** object contains a collection of sprites. A sprite is a rectangle that is rotated to face the viewer. When properly textured, a Sprite can realistically simulate complex objects with point or axis symmetry, such as clouds or trees, respectively, but with far less cost than if the objects were modeled with complex geometry.

A sprite is a rectangle defined by two corners that are translated by the Sprite's position. The +Z axis of the Sprite's coordinate system is always rotated to face the viewer. How this rotation is constrained defines the sprite mode. Sprite modes include

- AXIAL: The object coordinate +Y axis is constrained to the sprite axis, defined in object coordinates. This mode is typically used for roughly cylindrical objects like trees.
- POINT_EYE: The object coordinate +Y axis is constrained to the sprite axis, defined in eye coordinates. This mode is typically used to keep text upright on the screen.
- POINT_OBJECT: The object coordinate +X axis is constrained to the sprite axis, defined in object coordinates. This mode is typically used to keep clouds from rolling with the viewer.

Sprites may be given an overall normal for lighting calculations. Their color is taken from the current Material.

csSprite contains the following methods:

```
void        setBottomLeft(const csVec3f& bottomLeft);
void        getBottomLeft(csVec3f& bottomLeft);
void        setBottomLeft(csFloat v0, csFloat v1, csFloat v2);
void        getBottomLeft(csFloat *v0, csFloat *v1, csFloat *v2);

void        setTopRight(const csVec3f& topRight);
void        getTopRight(csVec3f& topRight);
void        setTopRight(csFloat v0, csFloat v1, csFloat v2);
void        getTopRight(csFloat *v0, csFloat *v1, csFloat *v2);

void        setPosition(const csVec3f& position);
void        getPosition(csVec3f& position);
void        setPosition(csFloat v0, csFloat v1, csFloat v2);
void        getPosition(csFloat *v0, csFloat *v1, csFloat *v2);

void        setNormal(const csVec3f& normal);
void        getNormal(csVec3f& normal);
void        setNormal(csFloat v0, csFloat v1, csFloat v2);
void        getNormal(csFloat *v0, csFloat *v1, csFloat *v2);

void        setAxis(const csVec3f& axis);
void        getAxis(csVec3f& axis);
void        setAxis(csFloat v0, csFloat v1, csFloat v2);
void        getAxis(csFloat *v0, csFloat *v1, csFloat *v2);

void        setMode(ModeEnum mode);
ModeEnum    getMode();
```

You can set the size of the sprites by setting the opposite corners of a sprite rectangle using **setBottomLeft()** and **setTopRight()** methods. **setPosition()** allows you to specify the lower-left corner coordinates of the sprites in the collection. **setNormal()** allows you to specify the orientation of all the sprites. **setAxis()** allows you to specify the axis around which the sprite orients itself. **setMode()** specifies the sprite mode; the valid values are enumerated in ModeEnum.

```
enum ModeEnum
    {
        AXIAL,
        POINT_EYE,
        POINT_WORLD,
    };
```

These values are explained earlier in this section.

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 colore.

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 4

Specifying the Appearance of Geometries

Two classes specify the appearance of a **csGeometry** object:

- csContext
- csAppearance

csContext provides the default appearance values for all of the shapes in a scene graph. **csAppearance** objects allow you to customize the appearance of shapes.

This chapter describes the appearance fields in **csContext** and **csAppearance**.

These are the sections in this chapter:

Specifying the Appearance of a Geometry

A **csContext** object contains all of the default appearance values necessary to render a shape; but it does not specify the shape to render. The appearance fields of a **csContext** object specify such things as the surface reflectance of a geometry (**csMaterial**), the texture that is applied to the geometry (**csTexture**), whether or not the geometry is transparent, and what shading model is used in the rendering.

csAppearance objects inherit all of their field values from **csContext** objects by default. These field values should be reset in a **csAppearance** object only if they change often.

Tip: 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.

csContext and csAppearance Differences

csAppearance objects have the same appearance fields and methods as **csContext**, except that **csAppearance** has **setInherit()**.

The **setInherit()** function specifies, by way of a bitmask, whether or not **csAppearance** fields inherit their values from the fields in a **csContext** object. For more information about **setInherit()**, see "Inheriting Appearance Values" on page 45.

The **getLightTarget()** method returns the ID of the geometry the light is pointed at.

All of the other appearance fields and methods in **csContext** are identical to those in **csAppearance**.

Since the appearance fields in **csContext** are identical to those in **csAppearance**, the description of the fields is presented only in the discussion of **csAppearance**.

Changing the Context

You can create multiple **csContext** objects, only one, however, 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 can define the appearance of a **csGeometry** object, for example, its texture, material, or color. Almost all of the fields in **csAppearance** are duplicates of the fields 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 and the more complex the database required to handle the values.

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 list of **set...()** methods are described in greater detail in the rest of this chapter:

```
void setInherit(const csBitMask& inherit);

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 three groups:

- The **setInherit()** method that you use to set the bitmask.
- Those methods containing the string “tex,” which 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 squeezed, expanded, or repeated to fit on the surface of a 3D object, such as a sphere. The squeezing 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.

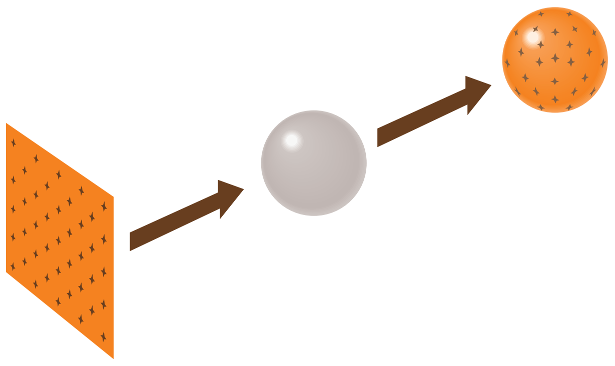

Figure 4-1 Applying a Texture to a Geometry

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 4-2.

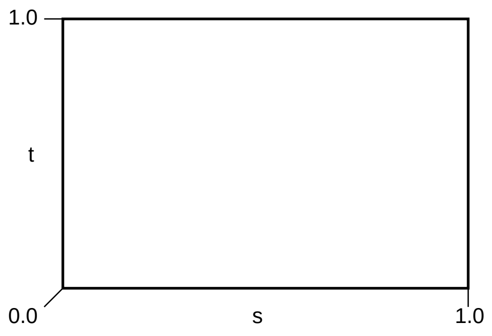

Figure 4-2 Texture Coordinates

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 NormalSet.

setTranspMode(), **setTranspEnable()**
to specify transparency.

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
```

csImage 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.

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

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 4-3.

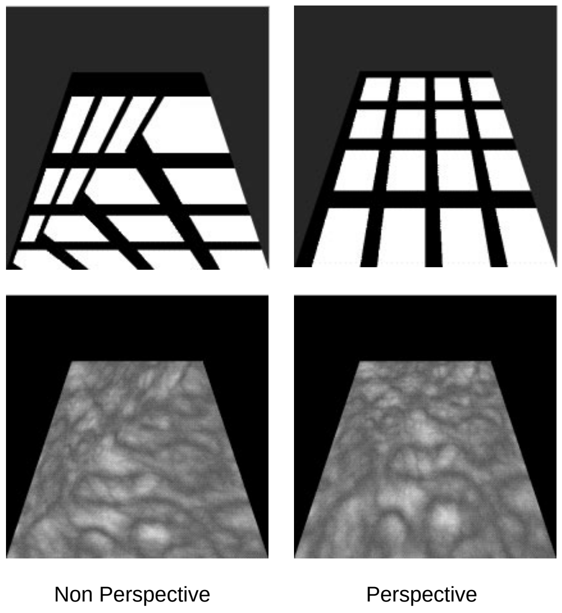

Figure 4-3 Non-Perspective and Perspective Modes

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 set the way in which texture colors are blended in 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, you might want to surround 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 as it 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 4-4.

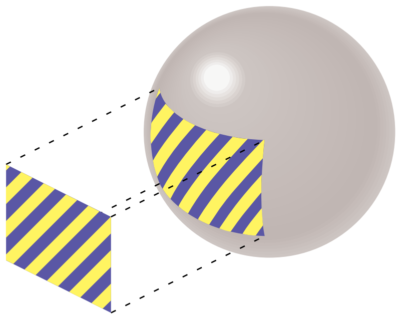

Figure 4-4 Texture Coordinate Function

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

Figure 4-5 shows how a texture plane is repeated across a geometry.

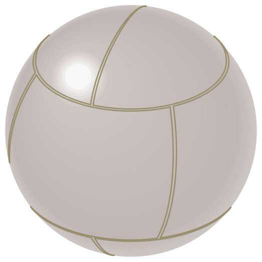

Figure 4-5 Repeated Texture on a Geometry

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.

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...()** functions.

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.

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.

Commonly, you use the **setTranspMode()** twice to specify FAST_TRANSP or NICE_TRANSP, and 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.

If you use NEVER_AFUNC, the incoming pixel is never rendered; this function has the effect of creating a totally transparent geometry. If you use LESS_AFUNC, the incoming pixel is rendered only if its alpha value is less than the reference value. If you use ALWAYS_AFUNC, the incoming pixel is always displayed; this function has the effect of creating an opaque geometry.

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” on page 59.
- “Group Nodes” on page 61.
- “Leaf Nodes” on page 61.
- “Setting the Values in Scene Graph Nodes” on page 63.

What Is a Node

Only nodes can be part of a scene graph. 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 5-1.

Table 5-1 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 9, “Lighting.”

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 5-1.

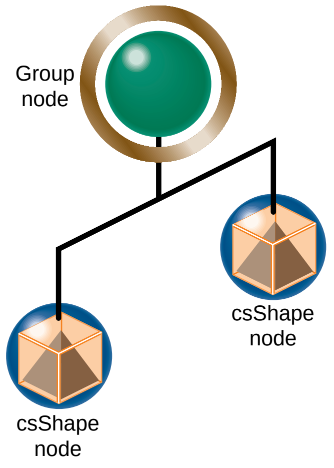

Figure 5-1 A Simple Grouping

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 8, “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.
- **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 13, “Optimizing Rendering.”
- **csTransform**—positions and orients a shape in the coordinate system of the parent node to **csTransform**. For more information, see Chapter 7, “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 9, “Lighting.”

The following section describes **csSwitch**.

Switching Between Nodes

The **csSwitch** node selects none, one, or all of its children, depending on the value of the argument in its constructor:

```
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.

Using csSwitch

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.

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 5-2.

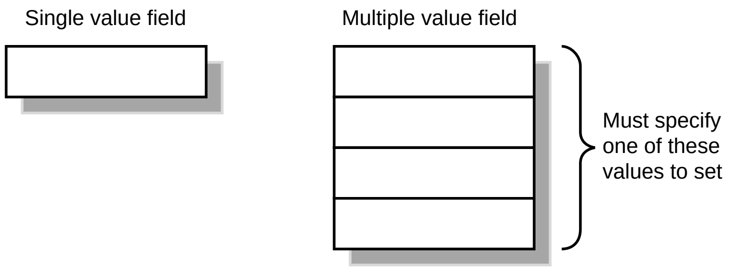

Figure 5-2 Setting Single and Multiple-Value Variables

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 6

Building a Scene Graph

A scene graph can be a single node or a hierarchy of nodes, as shown in Figure 6-1.

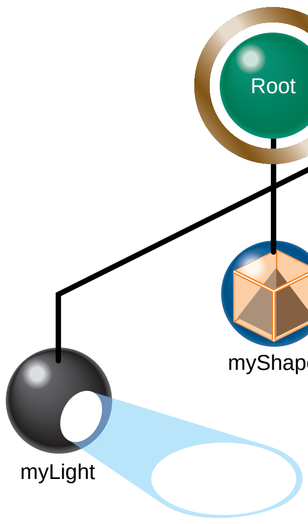

Figure 6-1 Scene Graph

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" on page 68.
- "Diagramming Scene Graphs" on page 70.
- "Altering Scene Graphs" on page 74.

- "Loading a VRML Scene Graph" on page 75.
- "Saving Scene Graphs" on page 76.
- "Troubleshooting Scene Graph Construction" on page 76.

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 6-2.

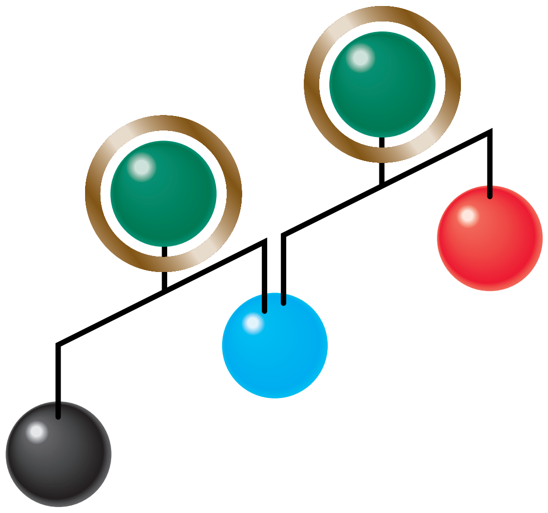

Figure 6-2 Multiple Root Nodes

Applying Actions to Root Nodes

An action applied to one of the roots would only traverse to 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 6-2, 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 6-1 is a simple scene graph.

Example 6-1 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 *myShape = new csShape;


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

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 6-3 shows a diagram representing the scene graph coded in Example 6-1.

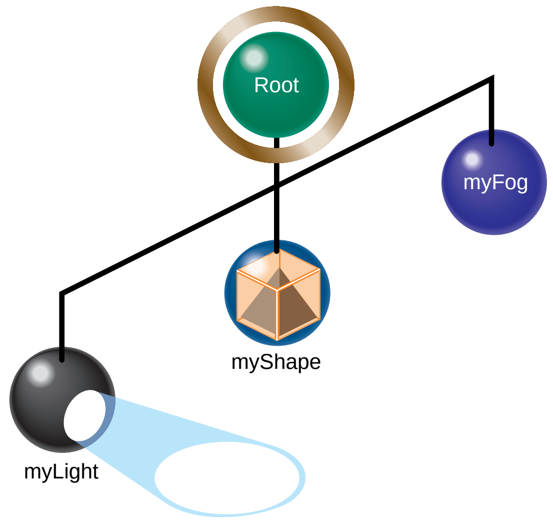

Figure 6-3 Simple Scene Graph

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 6-1. Remember, however, that actions may traverse these leaf nodes in any order since they all 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 6-4 shows that the two subgraphs, Molecules and Hydrogen bonds, are separate geometries that appear together in world space.

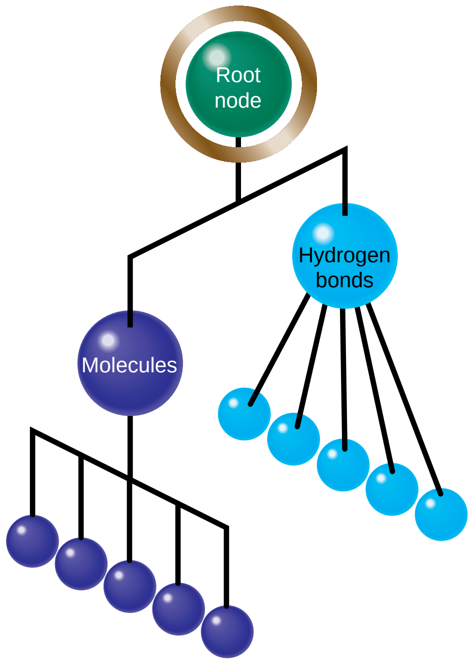

Figure 6-4 Two Sets of Data Rendered Differently

In the subgraph shown in Figure 6-5, you can see that the foot, leg, and torso nodes are parts which, when rendered together, display the lower half of a body.

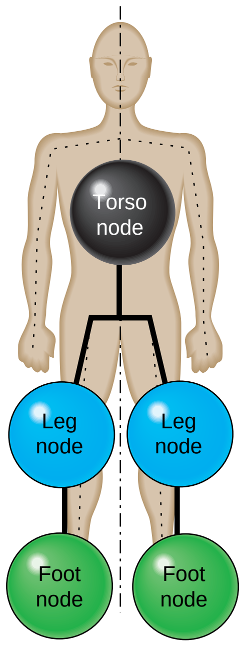

Figure 6-5 Torso Subgraph

Since 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 6-6.

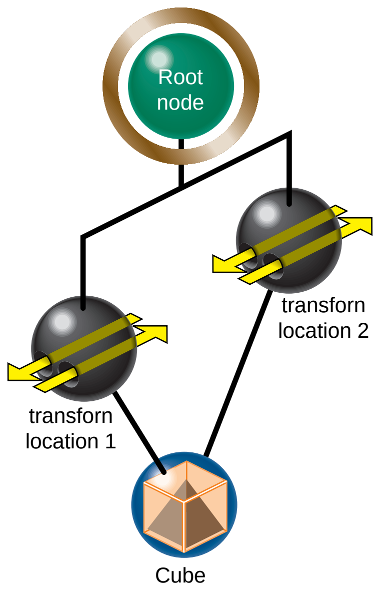

Figure 6-6 Showing the Same Geometry in Two Locations

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, it is always the case that the first node added to a group node is 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 6-2 shows a portion of *vrml.cxx*. This code shows how to load a VRML scene graph.

Example 6-2 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, might 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 Cosmo3d in-memory data structure to store. The method returns TRUE if the file is stored successfully, FALSE otherwise.

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 9, “Lighting” and Chapter 10, “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 7

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 96.

This chapter has the following sections:

- "Creating a Sense of Depth" on page 77.
- "Transforming Shapes to New Locations, Sizes, and Orientations" on page 78.

Creating a Sense of Depth

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

Overriding the Default Order of Layering Shapes

To override this layering effect, you can use the **csDepthFunc()** 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

For example, if you use 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. If you use 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. Choosing ALWAYS_DFUNC always displays the incoming pixel 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 7-1.

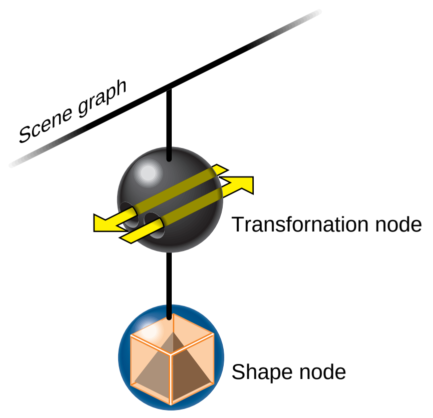

Figure 7-1 Placement of csTransform Nodes

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 7-2 shows a shape scaled by a factor of 2 in two different orientations, 0 and 45 degrees, respectively.

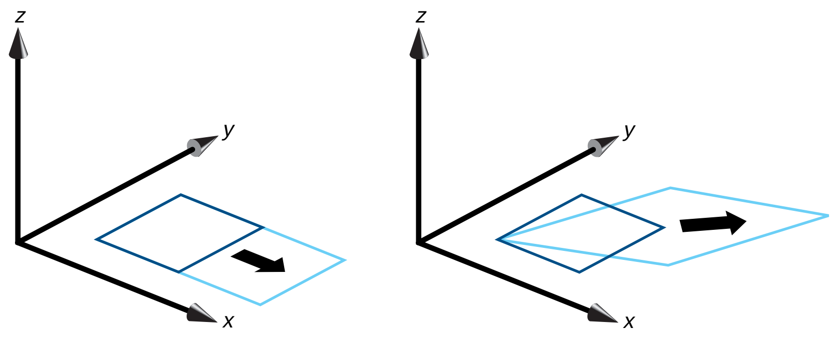

Figure 7-2 Scaling in Different Orientations

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 an image. 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 7-3.

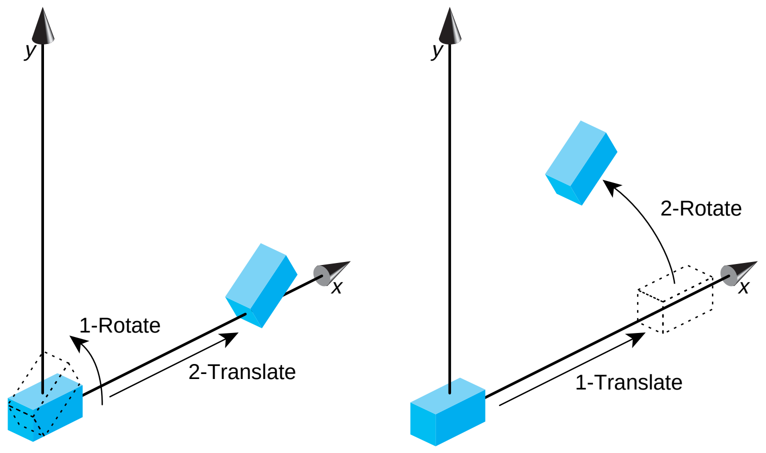

Figure 7-3 Order of Transformations

When you reverse the order of the transformations, the end result is different. Since the center of rotation is about 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.

Locating Transformation Nodes in Scene Graphs

A transformation node appears in a scene graph between a shape node and the remainder of the scene graph, as shown in Figure 7-4.

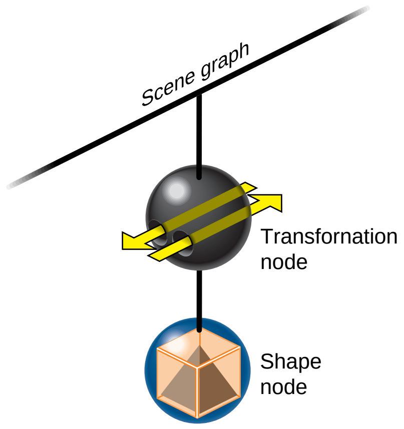

Figure 7-4 Placement of a Transformation Node

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. The next sections describe both approaches.

Chapter 8

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 119.
- **csCompileAction**, see “Compiling Part of a Scene Graph” on page 129.

Deriving from 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 14, “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 8-1 show one order in which an action might traverse the nodes in a scene graph.

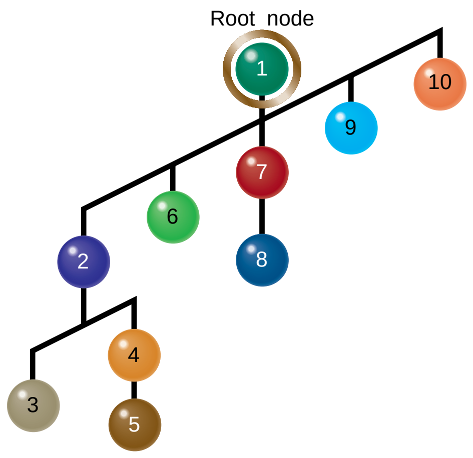

Figure 8-1 The Flow of an Action Through A Scene Graph

A top-down traversal means that a node can never affect a node “above” it. For example, node 3 in Figure 8-1 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 8-1, 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 9

Lighting

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**.

This chapter describes how to use lights, change the shadow modeling, and change the screen to one color.

This chapter contains the following sections:

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. 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 beam width of the light is set with the following method:

```
void setBeamWidth (csFloat beamWidth);
```

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 lay within the **csPointLight**’s shining radius are affected by the **csPointLight**. **csTransform** objects affect the location and shining radius of each **csPointLight**.

You 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);
```

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	where 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	where shade values are interpolated across primitive, geometric polygons. This option makes the primitive polygons look more like a curved surface.

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.

Chapter 10

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 95.
- "Using a Camera to View a Scene" on page 96.

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 10-1.

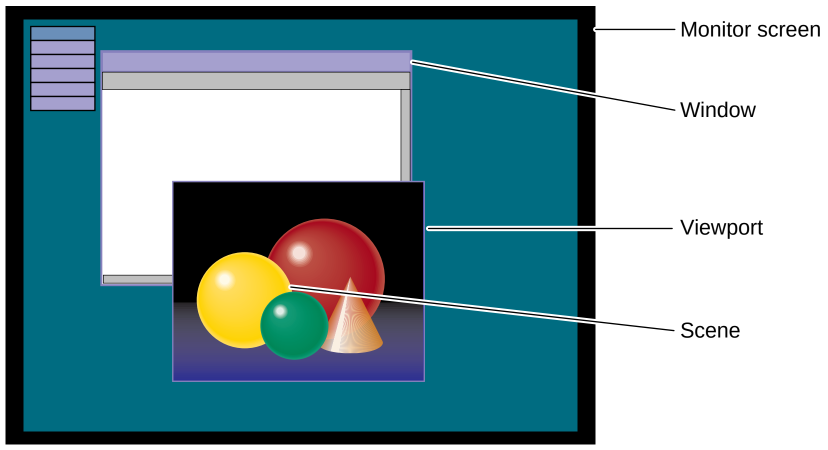

Figure 10-1 Viewport

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 dimensions 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 97.
- “csOrthoCamera” on page 99.
- “csPerspCamera” on page 99.

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 10-2.

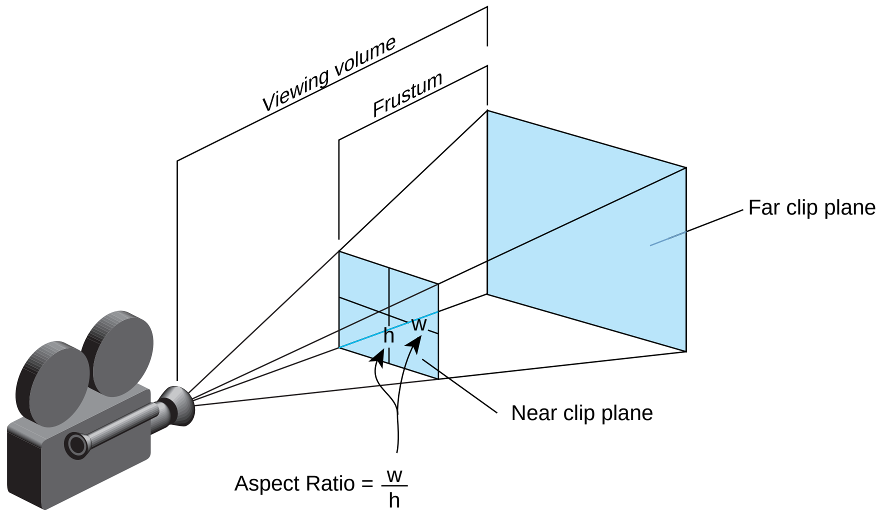

Figure 10-2 Aspect Ratio

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 10-3.

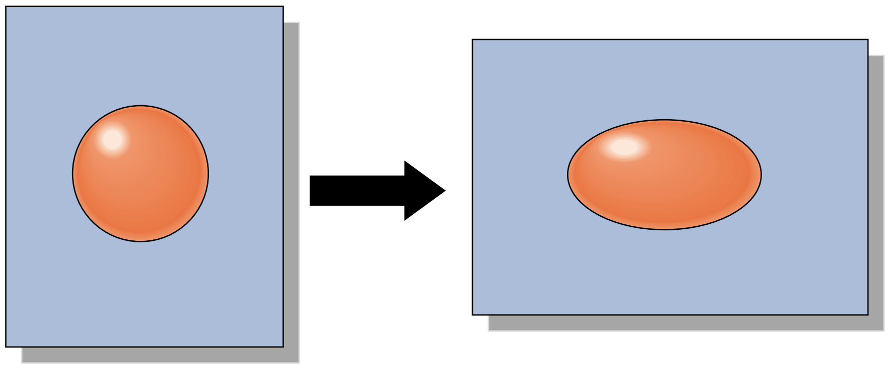

Figure 10-3 Changing the Window Without Changing the Image's Aspect

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 following fields set the orientation of a camera and the aspect mode of the image it takes.

```
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.

The following methods set the frustum:

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

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

You use the **setPosition()** methods 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.

The following sections describe the different cameras available in Cosmo 3D.

csOrthoCamera

csOrthoCamera defines an orthographic projection. An orthographic projection uses a parallelepiped (box) frustum. Unlike the frustum in Figure 10-2, 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're 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);
```

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 10-4.

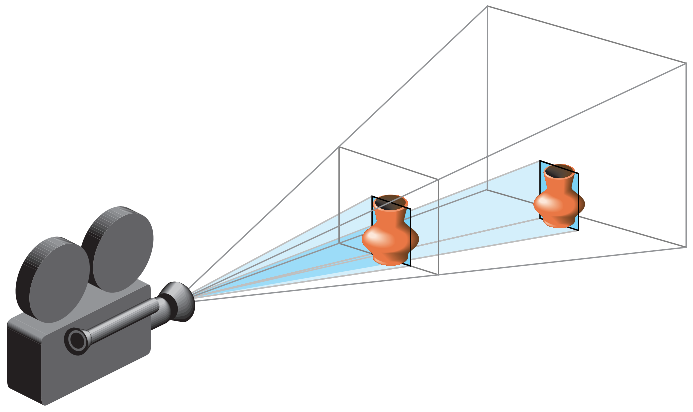

Figure 10-4 Perspective Explained

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.

You use the following **get()** fields to set the horizontal and vertical fields of view (FOV) values of the frustum.

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

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

Chapter 11

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, for example, attach an engine to the transform node of each wheel of the car to animate its rotation.

You might also tie the motion of one tire to the motion of all the other tires so that when one wheel moves, the others move. 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. You might also tie the motion of the car to the motion of the **csCamera** so the camera follows the car. Finally, you might cycle the car through a set of colors repeatedly.

A **csEngine** is a derivative of **csNode**, but usually it is not included as a node in a scene graph. It is, of course, included in your application that contains the scene graph being rendered.

Input and Output Fields

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

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. It's output might be moving 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 as its input two color specifications and the number of gradations you want between the two. The output for the engine could be an incremental change of colors between the beginning and ending color values.

Engine Terminology

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 11-1 illustrates these terms.

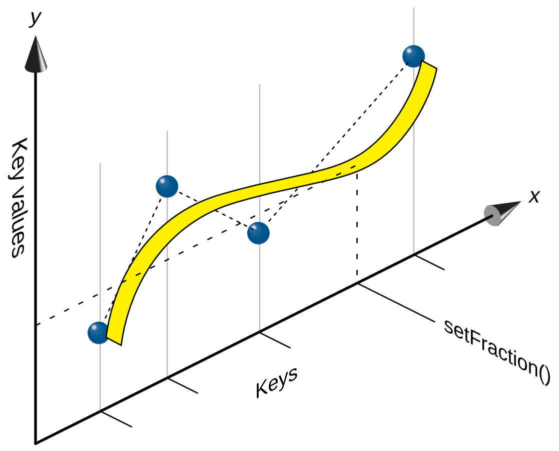

Figure 11-1 Engine Terminology

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:

```
csColorInterpolator* myEngine = new csColorInterpolator;
myEngine->connect(MYFIELD, yourEngine, YOURFIELD);
```

where *yourEngine* is also a **csEngine**. To make a color interpolator engine change a specific field in a shape, connect the engine to it as follows:

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

myEngine->getfield(VALUE)->connect(rose->getfield(DIFFUSE_COLOR);
// do a color cycle here
```

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

Engine Types

In general, you use interpolator engines to create the data used by other engines that change some feature of a shape.

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.
- **csInterpolator**—interpolate between keyframe values, selected from the key array by a floating point fraction, ranging from 0 to 1.
- **csColorInterpolator**—interpolates among a set of MFColor key values.
- **csCoordinateInterpolator**—linearly interpolates among a set of MFVec3f values.
- **csNormalInterpolator**—interpolates among a set of MFVec3f values.
- **csOrientationInterpolator**—interpolates among a set of MFRotation values.
- **csPositionInterpolator**—linearly interpolates among a set of MFVec3f values.
- **csScalarInterpolator**—linearly interpolates between a set of MFFloat values.
- **csSelectorEng**—selects one coordinate from an array input as its single output. Derived classes include **csSelectorEng3f** and **csSelectorEng4f**.

The following **csEngines** change some feature 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.**

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 11-2. The keys might be location, normal, or rotation values.

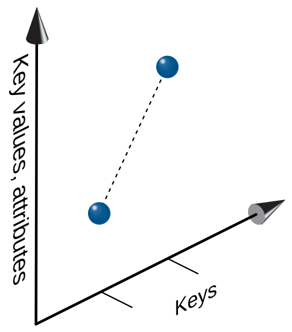

Figure 11-2 Keys and Key Values

Interpolator nodes are designed for linear, keyframed animation, that is, an interpolator node defines a piecewise linear function, **f(k)**, on the interval (-infinity, infinity). 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*.

csInterpolator Fields

The fields in **csInterpolator** include:

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

csMFFloat*          key();
```

For an explanation of **key()** and **setFraction()**, see, “Engine Terminology” on page 102.

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 11-3 shows two different splines that use the same key values, but are of different orders: piecewise linear and cubic.

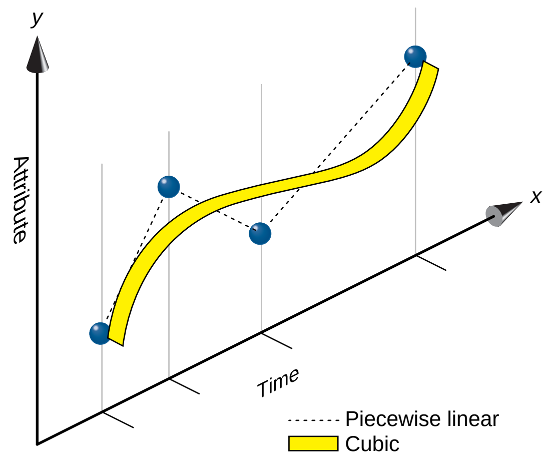

Figure 11-3 Spline

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

Keys and Key Values

In Figure 11-3, 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 non uniform. 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.

Different Orders of Splines

The matrix field in **csSpline** enables you to specify an infinite number of splines. Cosmo 3D, however, provides tokens for the following splines:

- piecewise linear
- quadratic
- cubic

Spline curves can be more or less smooth according to the order of the spline. The higher the order, the smoother the curve. Whereas a piecewise linear spline connects the input points with straight lines, the cubic spline performs a weighted average of the coordinates to create a smooth interpolation between the points. Smoothing the curve in this way produces smooth changes in location or attribute values.

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, “Engine Terminology” on page 102.

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

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

csColorInterpolator

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

csColorInterpolator contains the following field:

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

keyValue() is a 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.

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;
```

keyValue() contains the keys used for the coordinate interpolation.

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;
```

keyValue() contains the keys used for the coordinate interpolation.

csOrientationInterpolator

csOrientationInterpolator is identical to a **csNormalInterpolator** engine except that it considers an additional field: rotation. **csOrientationInterpolator** then interpolates between the quaternion values of X, Y, Z, and rotation for two unit vectors.

A **csOrientationInterpolator** interpolates between two orientations by computing the shortest path on the unit sphere between the two orientations. 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:

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

keyValue() contains the keys used for the coordinate interpolation.

csPositionInterpolator

csPositionInterpolator linearly interpolates between sets of **SFVec3f** values. 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:

```
MFVec3f keyValue[]
```

For an explanation of **keyValue()**, see, “Engine Terminology” on page 102.

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 field:

```
MFVec3f keyValue[]
```

For an explanation of **keyValue()**, see, “Engine Terminology” on page 102.

csSelectorEng

csSelectorEng selects one coordinate from in the input array as its output. **csSelectorEng** is an abstract class from which you subclass. Subclasses included in Cosmo 3D include **csSelectorEng3f** and **csSelectorEng4f**.

csSelectorEng contains the following field:

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

csSelectorEng3f and csSelectorEng4f

csSelectorEng3f and **csSelectorEng4f** are derived from **csSelectorEng**. The only difference between the parent and children classes is that the children classes restrict their input and output values to **csMFVec3f** and **csMFVec4f,** respectively.

Engines That Change Shapes

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

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 vec arrays, such as a **csMFVec3f**, according to the dimensions of the attribute data.

csMorphEng is an abstract class from which you subclass. Subclasses included in Cosmo 3D include **cscsMorphEng3f** 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() contains the number of keys through which the **csMorphEng** morphs.

index() enables you to index into a **csVec()** array. You pass into **index()** the number of the **csMFVec3f** set you want to access. You do this to access only that part of the array you want to use. For example, rather than downloading seven different versions of a clock, a file might contain a clock with a pendulum seven different positions. To display the face, the application woutd display the face and then index into the file to the correct pendulum position.

weights() sets the relative weight of each of the **csMFVec3f** sets in the **csVec()**.

csMorphEng3f and csMorphEng4f

csMorphEng3f and **csMorphEng4f** are derived from **csMorphEng**. The only difference between the parent and children classes is that the children classes restrict their input and output values to **csMFVec3f** and **csMFVec4f,** respectively.

Example 11-1, taken from the test program, *worm.cxx*, shows how to build a **csMorphEng3f** engine.

Example 11-1 Building a Morph Engine: the Worm

```
// Build morph engine
    MorphCoords = new csMorphEng3f;
    // “Neutral” is non-indexed and always has weight of 1
    MorphCoords->vecs()->setRange(0, NumRings*RingVerts, coords);
    MorphCoords->counts()->set(0, NumRings*RingVerts);
    MorphCoords->weights()->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->vecs()->set(k+NumRings*RingVerts, v);
            MorphCoords->indices()->set(k, k);
        }
        // targets are indexed
        MorphCoords->counts()->set(i+1, -RingVerts);
    }
    MorphCoords->connectOutput(csMorphVec3f::OUT_VECS,
                               SkelCoords, csTransformEng::VECS);
```

This engine moves the rings in the worm.

csTransformEng

csTransformEng is a **csEngine** that translates attribute sets consisting of points or vectors. A transform engine changes the orientation and location of a shape. The **csTransformEng** output is a single array which may be used as a **csGeoSet** attribute list, **csCoordSet3f**, **csNormalSet3f**, for example

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

csTransformEng is an abstract class from which you subclass. Subclasses included in Cosmo 3D include **csTransformEng3f**. **csTransformEng3f** restricts its input and output values to **csMFVec3f**.

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 and are transformed by the inverse transpose of the matrices in **matrix()**.

matrix() specifies the transformation matrix. The matrix array should supply a matrix for each attribute set in input.

count() lists the number of attributes in each input, attribute set.

Optionally, attribute sets may index into the output list if the index **csMFInt** field has a non-zero count. If so, the **input[i]** vector contributes to **output[index[i]]**.

Example 11-2 creates a sample **csTransformEng**.

Example 11-2 Creating a csTransform Engine

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

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

csTransformEng3f

csTransformEng3f is derived from **csTransformEng**. The only difference between the two is that **csTransformEng3f** provides **input()** and **output()** fields of type **csMFVec3f**.

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 Your Own Window

Instead of using the window provided by Cosmo 3D, **csWindow**, you can instead create your own window using X11 window code. In this case your application controls the **csContext**, **csEvent**, and X window, as shown in Figure 12-1.

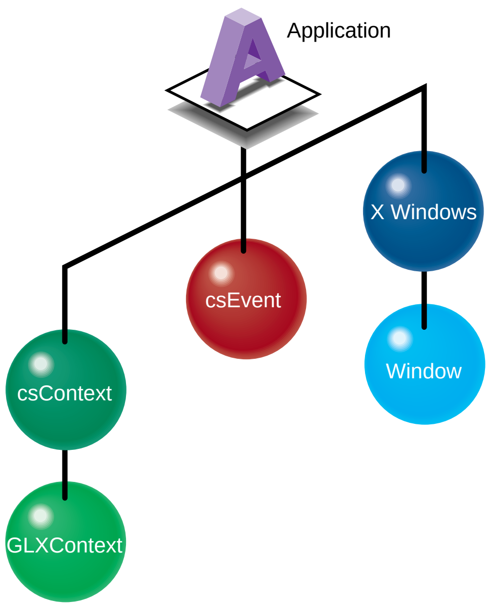

Figure 12-1 Creating Your Own Window

Creating a csWindow

All of the **csWindow** fields have default values. You may find that they satisfy the needs of your application. *cube.cxx*, for example, uses all the default values except that it provides a title for the window using the **setWindowTitle()** method.

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);
```

To show, hide, or iconify a window, use the following methods:

```
static void          iconifyWindow(void);
static void          showWindow(void);
static void          hideWindow(void);
```

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 **csWindow** object on top of the stack displays.

The following methods in **csWindow** manipulate the Window stack:

```
static csWindow*     getCurrent();
static void          makeCurrent(csWindow *win);
static void          popWindow(void);
static void          pushWindow(void);
static void          swapBuffers();
```

You place a **csWindow** object on top of the window stack using the **makeCurrent()** method. The **popWindow()** method removes the top **csWindow** object on the stack so that the object directly below it becomes the current window.

The **pushWindow()** method copies the **csWindow** object on the top of the stack and pushes the copy on the top of the stack, sinking the original **csWindow** object down one level.

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 the **swapBuffers()** method.

Handling User Input

User input to Cosmo 3D consists of

- mouse clicks
- window resize and update events
- key presses

To monitor user input, use the following methods:

```
static int          get(QueryEnum what);
static int          getDevice(QueryDeviceEnum what);
static int          getMouseX();
static int          getMouseY();
static unsigned int getMouseButtons();
```

QueryEnum covers a wide variety of window information, such as the window's height, width, and position. The **getDevice()** method determines which device the user event was generated by, such as the mouse, keyboard, or space ball. The remaining methods specify the mouse cursor location and which of the mouse's three buttons was pushed.

Handling Multiple Events

Cosmo 3D uses an event array to store events. You use the following methods to place the events in the array, and to erase all of the events in the array:

```
const csEventArray& getEvents();
void              resetEvents();
```

Handling Mouse Events

You handle mouse events using the following procedure.

1. Use **getDevice()** to get a handle to a device and determine which, if any devices, received user input.
2. Use the **getMouseButtons()** method to determine which mouse button was pressed. The return value (MouseButtonEnum) for the method specifies whether the left, middle, or right mouse button was pushed.

3. Use the **getMouseX()** and the **getMouseY()** methods to determine the location of the mouse cursor when the mouse button was pressed.

Selecting Screen Objects

Cosmo 3D enables the selection of screen objects in the following ways:

- **csIsectAction**—selects the shape closest to the camera based on a ray.
- **csCamera::pick()**—selects the shape closest to the camera based on window coordinates.

These mechanisms use the direction of the camera and its proximity to a screen objects to select one.

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 **csSeg** ray that emanates from the camera position as shown in Figure 12-2.

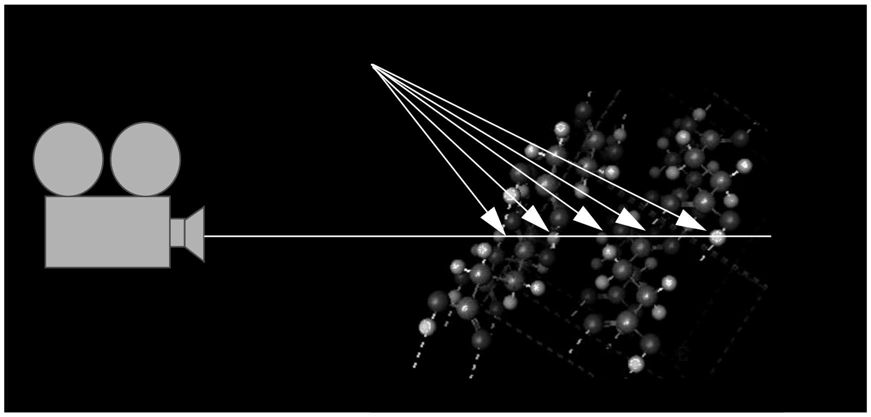

Figure 12-2 Ray Pick Action

The shape closest to the camera and intersected by the line segment is recorded in a **csHit** object. For information about **csHit**, see "Storing Selected Screen Objects" on page 120.

Using Pick()

csCamera::pick() uses window and viewpoint coordinates to select the screen object closest to the ray connecting the camera to the point. You might supply the window coordinates using the mouse methods; see "Handling Mouse Events" on page 118.

When you supply **pick()** with window coordinates, the method internally uses **csContext::getViewpoint()** 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 120.

For information about **csContext::getViewpoint()**, see "Setting the Screen Display of the Scene" on page 95.

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 of 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.

Chapter 13

Optimizing Rendering

One of the greatest challenges you face as a developer after you create an application is optimizing its performance.This chapter describes the Cosmo 3D nodes and programming techniques that can help optimize your application's performance.

For more information about performance tools, see the *OpenGL Optimizer Programming Guide*.

These are the sections in this chapter:

- "Reducing the Number of Vertices Rendered" on page 123.
- "Performance Programming Techniques" on page 127.

Reducing the Number of Vertices Rendered

The more vertices calculated and rendered, the slower the application's performance. To the extent that you can reduce calculations and rendering, you can improve your applications 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 123.
- "Level of Detail Reduced for Performance" on page 125.
- "Culling the View Frustum" on page 124.

Face Culling

When a 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.

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 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** children 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. This 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 13-1.

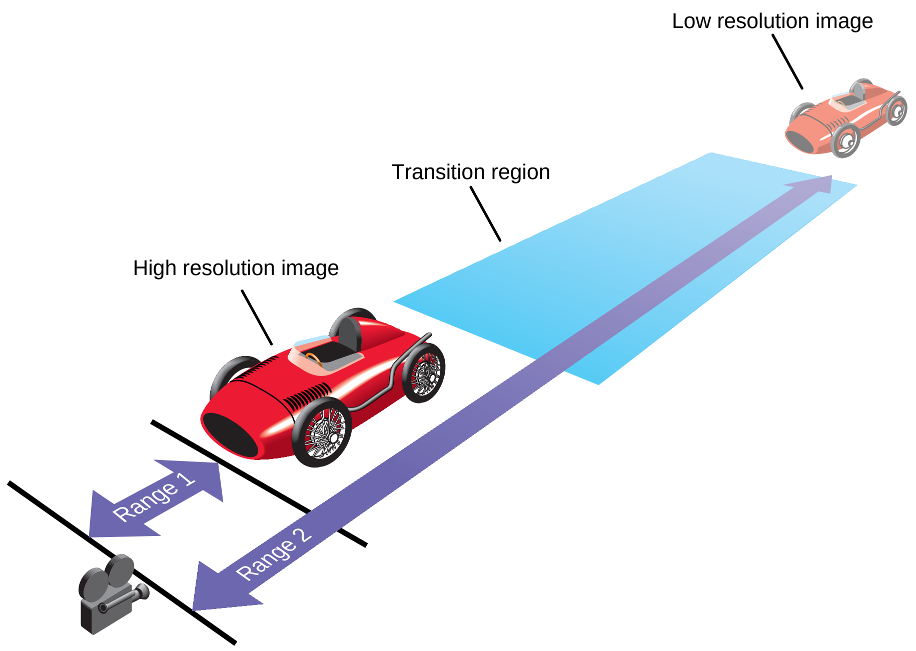

Figure 13-1 csLOD Ranges

Performance Programming Techniques

The following sections provided programming tips for improving the performance of your application:

- "Minimize Use of csAppearance Fields" on page 128.
- "Minimize Use of csAppearance Modes" on page 128.
- "Indexing csGeoSet Attributes" on page 128.
- "Setting the Transformation Matrix Directly" on page 128.
- "Compiling Part of a Scene Graph" on page 129.

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 acted upon.

To maximize performance, set **csContext** fields to values that satisfy a majority of the shapes in a scene. 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 the performance if your application. 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 30.

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 13-2. 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 the 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);
```

where *node_name* is the name of the node below which you want to precompile.

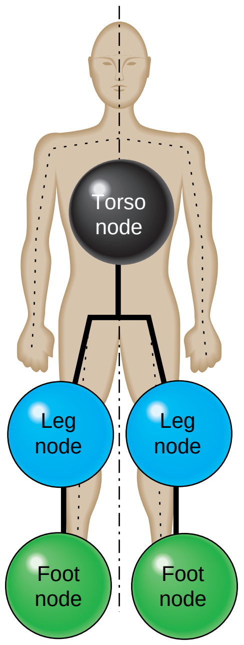

Figure 13-2 Arranging Scene Graph Nodes

Chapter 14

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" on page 131.
- "How to Play a Sound File" on page 137.
- "Specifying Audio Files" on page 139.
- "Playing Sound in Immediate Mode" on page 141.

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, either include the **csSound** object in the **csShape** node or 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 14-1.

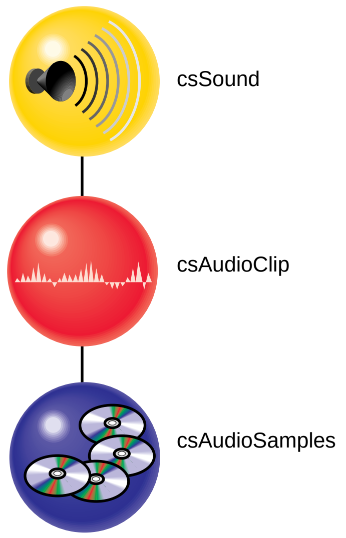

Figure 14-1 Sound Classes

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 137. For more information about **csAudioSamples**, see "Specifying Audio Files" on page 139.

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 137.

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.

Locating the sound source in a scene is as easy as passing 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 14-2.

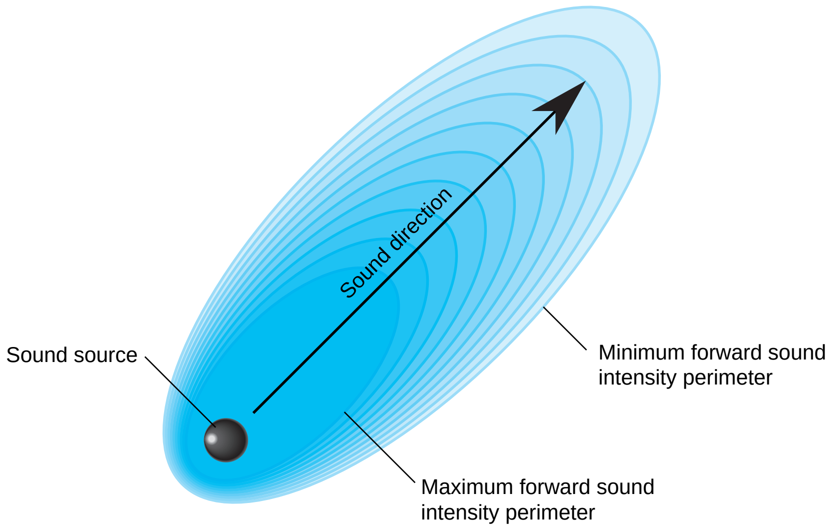
Sound direction
Sound source
Minimum forward sound intensity perimeter
Maximum forward sound intensity perimeter

Figure 14-2 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 14-2, 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 14-3.

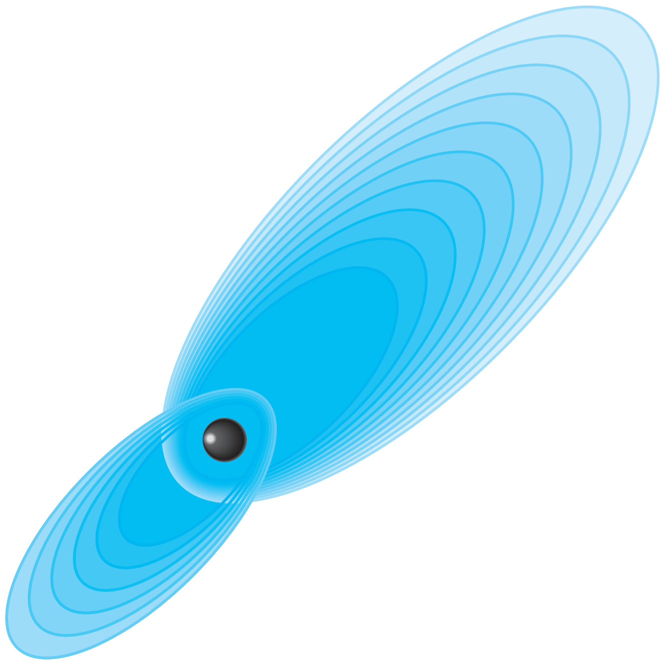

Figure 14-3 Forward and Reverse Sound Propagation

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.

csSound Methods

The **csSound** node provides virtual methods to handle **csDrawAction** and **csSoundAction** actions. The **csSound** node also provides the **fieldChanged()** method that specifies which **csSound** fields, if any, have changed since an action was last applied. If none of the fields changed, the fields in the node should not be evaluated. You might also create a method that keeps track of which fields changed so that only those fields are reevaluated.

The **csSound** methods are as follows:

```
virtual void fieldChanged(short fieldId);

virtual void draw(csDrawAction *da);
virtual void draw();

virtual void sound();
virtual void sound(csSoundAction *sa);
int sound(char *dst, int nFrames, float frequencyFactor,
    float intensityFactor, int numDstChannels);
```

Because these methods are virtual, you must provide your own implementation of them in your application.

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 14-1 describes how these fields are used.

Table 14-1 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.
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 14-1 sets all of the fields in a **csAudioSamples** node.

Example 14-1 Setting the Fields in an 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”0;
```

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 14-2 describes how these fields are used.

Table 14-2 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.

Table 14-2 (continued) Fields of csSoundSamples

Field	Description
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 14-2 sets all of the fields in a **csAudioSamples** node.

Example 14-2 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

- Open and close audio ports
- Update and clean the sound buffer
- Provide a virtual method to detect field changes
- Return the number of sound players available
- Specify the sound fiile to be played
- Locate the microphone (viewer) in the virtual world
- Locate the sound source in the virtual world
- Simulate a doppler shift for a moving viewer
- Simulate a doppler shift for a moving sound source
- Set the maximum and minimum sound levels for a sound source as well as the current intensity of the sound
- Allow you to set up a buffer size and queue to play sound files sequentially

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 s, size_t nbytes);
```

getData() returns the data.

getSize() returns the size of the data.

new() specifies the number, *s*, of bytes, of size *nbytes*, allocated.

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 Vec2fs.
- csVec3fArray—An array of Vec3fs.
- csVec3sArray—An array of Vec3ss.
- csVec4fArray—An array of Vec4fs.

- csMatrix4fArray—An array of Matrix4f.
- 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 references.
- 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 specify a specific array object to manipulate, use the following method:

```
<type>*      getArray() const;
```

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);
```

The first argument, *i*, is the position of the value in the array. The second argument, *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[]);
```

The first argument, *i*, is the position in the array where you want to begin setting values. The second argument, *count*, is the number of array elements you want to set. The third argument, *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);
```

To add a value after the last value in an array, to insert a value at a specified index location, to replace one value with another, or to remove a specific value or a value located at a specific index, use the **append()**, **insert()**, **replace()**, **remove()**, or **removeIndex()** methods, respectively. The **remove()** method removes the first value it finds that matches its argument.

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 System.out using the **write()** method.

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.
- 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**, **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 **csVec2f** 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 csVec2f &v);
void        get(csVec4f &v) const;
```

For the **csVec2f** class, the third and fourth argument members are set to zero in each of these methods. For the **csVec3f** class, just the last argument is set to zero.

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        xform(const csVec4f& v, const csMatrix4f& m);
```

The methods have the following functionality:

- almostEqual()—returns TRUE if the values are within .99 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()—square
- normalize()—makes the vector orthogonal to its original direction.
- distance()—determines the distance between two vectors.
- xform()—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) {...}
```

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 three ways to transform a **csVec3f** when passing a **csMatrix4f** into **xform()**:

- A **csVec3f** transform vector treats the **csVec3f** like a vector, as if it were a **csVec4f** with 0.0f stored as the fourth element. This is particularly useful for surface normals.
- A **csVec3f**.transform point treats the **csVec3f** like a point, as if it was a **csVec4f** with 1.0f stored as the fourth element, but does not perform perspective division. This is most useful for non-projective transformations.
- A **csVec3f**.transform point treats the **csVec3f** like a point, as if it was a **csVec4f** with 1.0f stored as the fourth element, but does perform the perspective division. This is useful for projective transformations such as perspective projection.

Bounding Volumes

Bounding volumes provide an efficient means of determining which shapes are in and out of the view frustum. The bounding volume for most shapes is a sphere, as shown in Figure A-1.

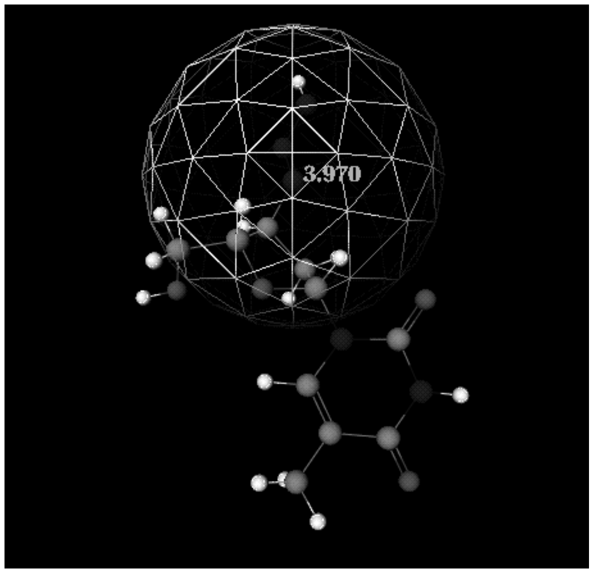

Figure A-1 Bounding Sphere

Cosmo 3D provides several bounding shapes as well as an abstract class from which you can derive your own bounding shapes. You use the bounding shape that most closely resembles the object being enclosed, for example, use a **csSphereBound** with spherical objects.

- 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 is the abstract base class from which bounding objects are derived. Methods are provided 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.

Cosmo 3D provides two bounding shape classes: **csBoxBound** and **csSphereBound**. Most shapes use a spherical bounding volume. Box-like shapes, however, such as csCube, use box-shaped bounding volumes.

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.

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 additional methods that allow you to create an enumeration by pairing an integer with a string, and to create an alias for the field name.

```
void                addEnum(int e, const char *str);
void                addAlias(const char *alias);
```

The following virtual fields instantiate, reset, write, and determine whether or not a specified container is the default value.

```
virtual csField*    instantiate(csContainer *p) = 0;
virtual void        reset(csContainer *p) = 0;
virtual void        write(csOutput *out, csContainer *cnt) = 0;
virtual void        write(csOutput *out);
virtual csBool      isDefault(csContainer *cnt) = 0;
```

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, for example, you might like to begin reading data. 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, you might read 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 in 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 single-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 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[]);
```

The first argument, *i*, is the position in the array where you want to begin setting values. The second argument, *count*, is the number of array elements you want to set. The third argument, *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 distance between a point off a plane and a point on a plane.
- Determine whether or not it intersects with a shape.
- Construct a plane from three points.
- Construct the normal to the plane 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.

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 007-3445-001.

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: 415-965-0964
- 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