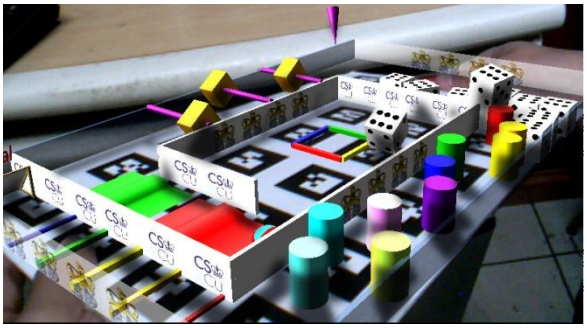
GOBLIN XNA USER MANUAL

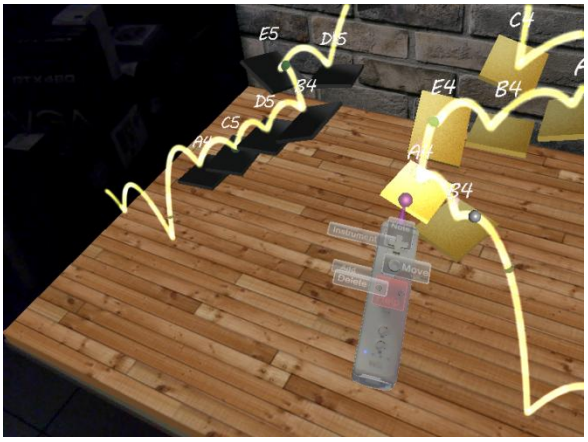
VERSION 4.0

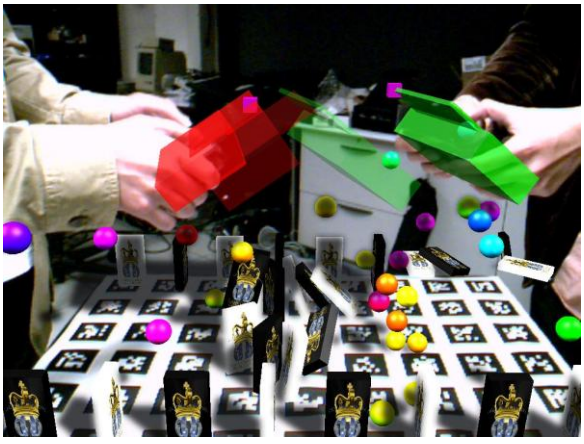
Ohan Oda
Colin MacAllister
Steven K. Feiner

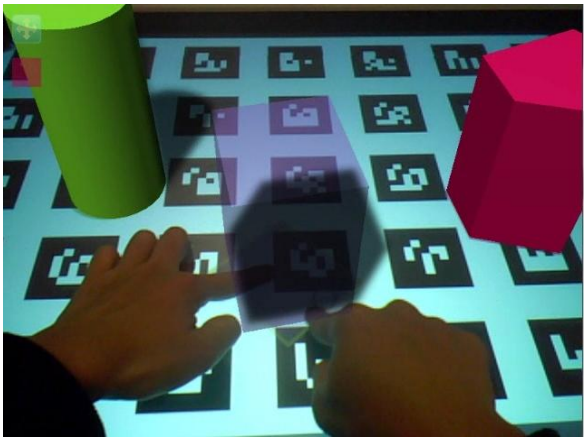
Columbia University
Department of Computer Science
New York, NY 10027



February 2, 2012







CONTENTS

INTRODUCTION

Goblin XNA is an open-source platform for building 3D user interfaces, including mobile augmented reality and virtual reality, with an emphasis on games. It is available for free download at http://goblinxna.codeplex.com under a BSD license, and is written in C# on top of Microsoft XNA Game Studio 4.0. Goblin XNA includes a scene graph to support 3D scene manipulation and rendering, mixing real and virtual imagery. 3DOF (three-degrees-of-freedom) orientation tracking and 6DOF (six-degrees-of-freedom) position and orientation tracking is accomplished using the VTT Research Centre ALVAR 6DOF marker-based camera tracking package with DirectShow, OpenCV or PGRFly (for Point Grey cameras), 3DOF and 6DOF InterSense hybrid trackers, and the 3DOF Vuzix iWear VR920 and Wrap Tracker 6TC (currently for 3DOF only) orientation trackers. Physics is supported through the Newton Game Dynamics 1.53 library and Havok Physics, and networking through the Lidgren library. Goblin XNA also includes a 2D GUI system to allow the creation of classical 2D interaction components.

Like all programs written using XNA Game Studio, Goblin XNA programs are developed with Microsoft Visual Studio.

Development of Goblin XNA was funded in part by a generous gift from Microsoft Research.

This material is also based in part upon work supported by the National Science Foundation under Grant No. 0905569. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Licensing information for Goblin XNA is detailed in *Goblin XNA License.rtf*, included in the Goblin XNA release.

GETTING STARTED

Begin by downloading all necessary dependencies and Goblin XNA itself, as described in the separate Installation Guide (*Installation Guide.pdf*), and set up everything as instructed. Once the development environment is properly installed, you can start either from one of the project templates in the *tutorials* folder or from scratch. First, we will describe how to start using the **Tutorial 1** project, and then how to start from scratch by creating a new project.

USING A PROJECT TEMPLATE

1. Open up the *Tutorials.sln* file in the *GoblinXNAv4.0\tutorials* directory by double-clicking it.
2. You will see fourteen tutorials. If 'Tutorial1—Getting Started' is not selected as a startup project, then right click on the 'Tutorial1—Getting Started' project file in the Solution Explorer pane, and select 'Set as StartUp Project'.
3. Build the solution by clicking the 'Build' toolbar button on the top, or by pressing the F6 key.[1]
4. If the build succeeds, you will see a 'Build Successful' message on the status bar at the bottom of the window. Now you are ready to run the project!
5. Run the project by clicking on the 'Debug' toolbar button on the top, and select either 'Start Debugging' (F5) or 'Start Without Debugging' (Ctrl + F5), depending on whether you want to see debugging information.
6. You should see a window pop up, inside of which will be a shaded 3D sphere model overlaid with the string "Hello World".
7. Double-click *Tutorial1.cs* in the Solution Explorer pane to view its code.

[1] You may notice that Visual Studio tries to build all of the projects in your solution, which slows down program start up when you run the project. This will not be very noticeable if you have only one project in your solution; however, if you have multiple projects in your solution, which is the case for the GoblinXNA tutorials, it will take some time before you see the program start.

One way to make your program start up faster (in Visual Studio 2010 Professional Edition, but not in Visual C# Express Edition) is to change one of the build and run options. Select the 'Tool' toolbar button, and select 'Options' at the bottom from the drop-down list. Expand the 'Projects and Solutions' section, and select 'Build and Run'. Make sure that the checkbox 'Only build startup projects and dependencies on Run' is checked.

Also, note that you do not need to execute 'Build' whenever you change your code. You can simply 'Run' or 'Start Without Debugging' your modified program, and Visual Studio will automatically build it for you before it runs.

FROM SCRATCH

1. Open Visual Studio 2010, and create a new project (select the 'New' menu item from the 'File' menu, and then select 'Project…').
2. Under the 'Visual C#' project type, select 'XNA Game Studio 4.0'. Then select 'Windows Game (4.0)'. Click the 'OK' button after specifying the project name and project folder.
3. You should see that the project has been created. The automatically created source code and other files will appear in the solution explorer window. Right-click on the 'References' section, and select 'Add Reference…' Change the selected tab to 'Browse', and then locate the *GoblinXNA.dll* you generated in the *GoblinXNAv4.0\bin* directory, as instructed in *Installation Guide.pdf*. Since managed dlls referenced by GoblinXNA will be automatically copied to the *GoblinXNAv4.0\bin* directory, you do not need to add them to your 'References' section unless you need to access specific APIs implemented in those managed dlls. If other managed dlls are not in the same directory as *GoblinXNA.dll*, then you will need to add them to your 'References' section.
4. Now, you are ready to use the Goblin XNA framework in your project. Note that if you want to use the physics engine, you will need to either copy *Newton.dll* (or *HavokWrapper.dll*) to your bin directory or add *Newton.dll* (or *HavokWrapper.dll*) to your project and set the property option 'Copy to Output Directory' to 'Copy if newer'. If you want to use the ALVAR marker tracker library, you will need to do the same steps for *ALVARWrapper.dll*, *alvar150.dll*, *alvarplatform150.dll*, *cv100.dll*, *cvaux100.dll*, *cvcam100.dll*, *cxcore100.dll*, and *highgui100.dll*, and the marker configuration file (a *.txt* or *.xml* file for ALVAR). These dlls *cannot* be added to the 'References' section because they are not managed.

HELLO, WORLD

To illustrate the basic setup of a Goblin XNA project we will walk through a simple "Hello, World" example. The code for this example can also be found in Tutorial 1. The following information pertains to Microsoft Visual Studio 2010 and XNA Game Studio 4.0. However, Microsoft Visual C# 2010 Express Edition uses procedures that are very similar or identical to our example setup.

INITIALIZING YOUR PROJECT

1. Launch Visual Studio and choose 'File'… 'New'… 'Project'.
2. In the left-hand 'Project Types' list, under 'Visual C#' choose 'XNA Game Studio 4.0'.
3. To the right is the 'Templates' area. Choose 'Windows Game (4.0)'.
4. In the 'Name' field enter a name for your project (e.g., "HelloWorld").
5. Your projects 'Location' can be anywhere you decide. The default is *C:\Documents and Settings\YourUserName\My Documents\Visual Studio 2010\Projects* (on a Windows XP machine); most modern operating systems also recommend user-specific locations. This is fine, especially if you work with Visual Studio often and store your projects here. Alternatively, you might wish to save your Goblin projects to the *GoblinXNAv4.0\projects* directory that was created when you downloaded Goblin XNA.
6. Click 'OK'.

THE DEFAULT ENVIRONMENT (XNA'S DNA)

After project creation, Visual Studio opens your project in the default view:

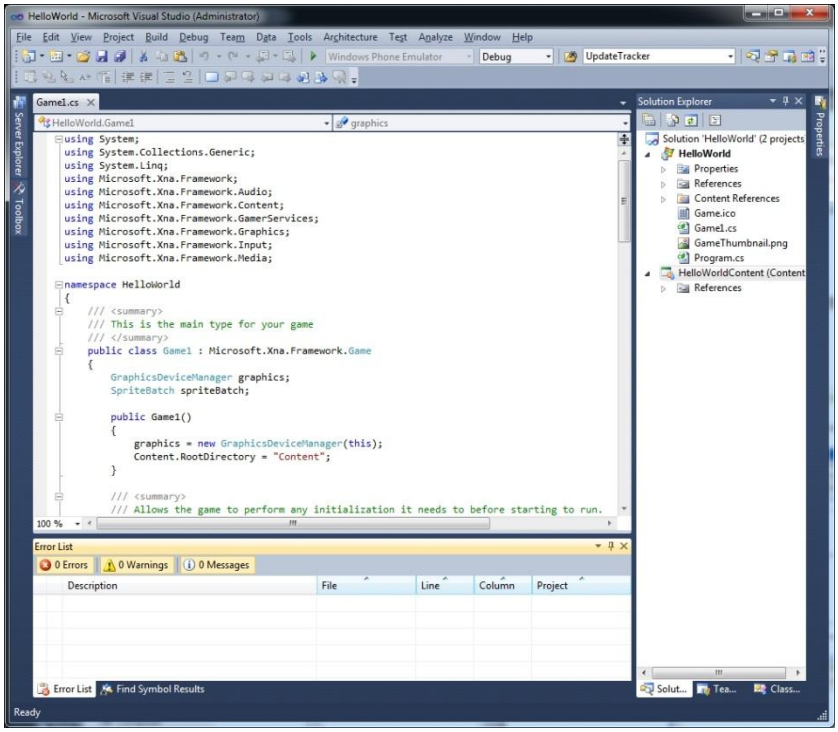


Figure 1: The default project view.

Your main code file will be created with the name *Game1.cs*. You can change this at any time by selecting the file in the Solution Explorer, then clicking it once (the same way you would change a filename in the Windows File Explorer.) You will then receive a message asking you if you wish to change all references to this file. Most likely, you

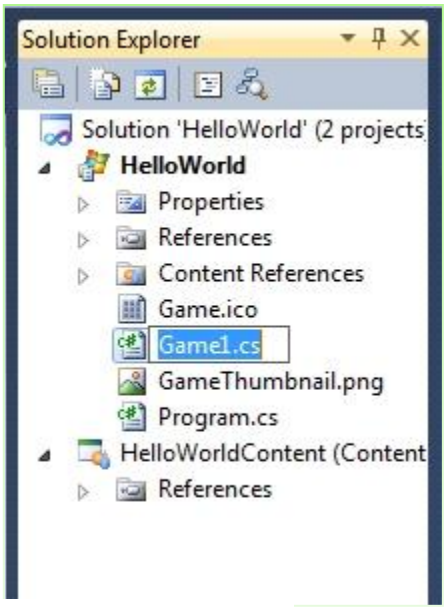


Figure 2: Enlarged view

will wish to do this.

BASIC ELEMENTS OF AN XNA GAME STUDIO APPLICATION

In your main code window, you will see that XNA Game Studio set up a number of `using` includes at the top of the file, which were created when your project was first initialized from the 'XNA Game Studio' template:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
```

In addition to a constructor method, five required methods were declared:

```
Initialize()
LoadContent()
UnloadContent()
Update()
Draw()
```

Two class objects were automatically declared before the constructor: the `GraphicsDeviceManager` and the `SpriteBatch`. Microsoft describes the `GraphicsDeviceManager` as the class that "handles the configuration and management of the graphics device." It allows your application to interact with and set the features of your graphics hardware. Often it is the `GraphicsDevice` property of the class that is queried or manipulated. More information about what this class does can be found at http://msdn.microsoft.com/en-us/library/bb194916.aspx#GraphicsDevice.

To handle 2D graphics, such as heads-up displays and scores, XNA Game Studio provides the `SpriteBatch` class. More information about this class and its use in creating 2D graphics can be found on the MSDN site, http://msdn.microsoft.com/en-us/library/bb203919.aspx.

You will notice that the constructor for your main class (in our example, `Game1`) contains the instantiation of `GraphicsDeviceManager` and sets a new property:

```
Content.RootDirectory = "Content";
```

The value of `Content` refers to the subdirectory of the solution that contains various assets you will use in your project. For example, you may require fonts for writing text

or sound files for playing audio.[2] Files such as these should be kept here, possibly in appropriately named subdirectories. Visual Studio provides this folder in the Solution Explorer. As you can see, this *Content* directory is created with a number of dynamically linked libraries that XNA Game Studio uses. So, the *Content* directory can contain both physical files and links to files located elsewhere in the system.

BASIC APPLICATION METHODS

You will notice that there are a number of methods pre-created by Visual Studio, including `Initialize()`, `LoadContent()`, `UnloadContent()`, `Update()`, and `Draw()`. These methods fill vital roles in an application, although we will not be using all of them in our "Hello, World" example. Descriptions of these methods can be found in comments above their signatures.

The `Initialize()` method is usually where you put non-graphical initialization code for your application. It is also where you call `base.initialize()`, which initializes other components, creates the `GraphicsDevice`, and calls the `LoadContent()` method. A more in-depth look at the correct order in which to do things in the `Initialize()` method can be found at http://nickgravelyn.com/2008/11/life-of-an-xna-game/.

[2] Note that the files included in your *Content* directory must be in formats that are recognized by XNA Game Studio.

CODING "HELLO WORLD"

At this point, we actually have a working application that can be built and run. Our final "Hello, World" example will be no "World of Warcraft", but it will be better than this:

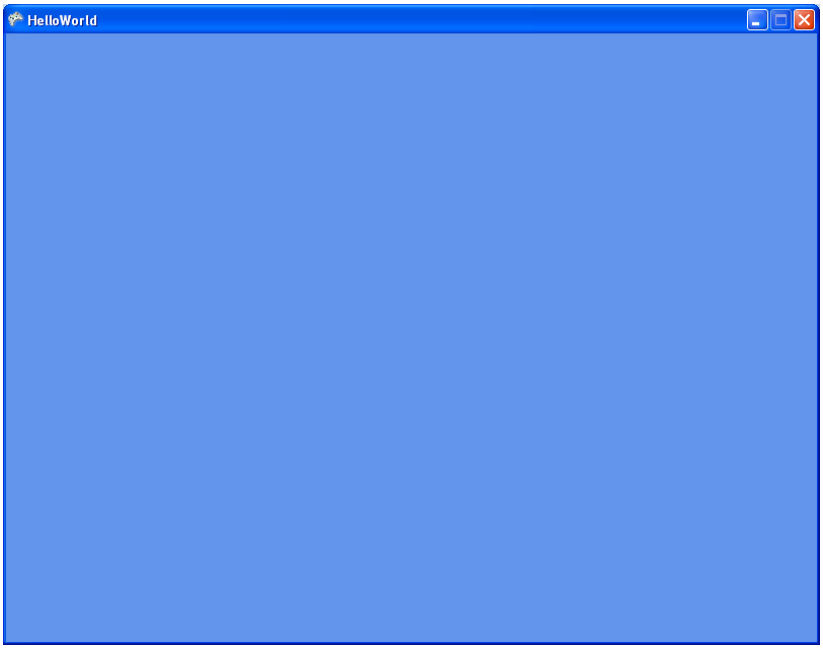


Figure 3: A blank XNA application.

First, to set up our Goblin XNA program, we will need a few more includes:

```
using GoblinXNA;
using GoblinXNA.Graphics;
using GoblinXNA.SceneGraph;
using Model = GoblinXNA.Graphics.Model;
using GoblinXNA.Graphics.Geometry;
using GoblinXNA.Device.Generic;
using GoblinXNA.UI.UI2D;
```

(More information about what these packages provide can be found later in this manual.)

CLASS OBJECTS

First we will create two class objects, one representing the scene graph and one the font we will use. A *scene graph* specifies the logical structure of a graphics scene, conventionally represented as a hierarchical directed acyclic graph (DAG) of scene graph nodes. A Goblin XNA scene graph contains nodes that represent and organize the visual and audio components of a scene. More information about the nodes that

can be used in a Goblin XNA scene graph can be found later in this manual. The code we use to create the scene graph is

```
Scene scene;
```

To create the text in our application, we need to specify a font. We do that by creating a `SpriteFont` object, which can be referenced when we use the `UI2DRenderer`. It will be loaded in the `LoadContent()` method by the following code:

```
/// LoadContent will be called once per game and is
/// the place to load all of your content.
protected override void LoadContent() {
     textFont = Content.Load<SpriteFont>("Sample");
}
```

In this case, the string "`Sample`" refers to a file that has been referred to in the project's *Content* directory. By putting a font file named `Sample.spritefont` (or a reference to such a file) in the *Content* directory, it becomes available to us within the code. If there were many such fonts being loaded, it might be better to create a *Fonts* directory within *Content* and place all fonts there.

Content can be unloaded by using the `UnloadContent` method, which is inherited from the Game class.

```
/// UnloadContent will be called once per game and is
/// the place to unload all content.

protected override void UnloadContent() {
     Content.Unload();
}
```

INITIALIZE

The next few lines perform a variety of tasks, including determining whether or not to show the mouse cursor, and what color to set the default background. Again, `Initialize()` is customarily used to do work other than the loading of external models.

```
base.Initialize();

// Display the mouse cursor
this.IsMouseVisible = true;

// Initialize the GoblinXNA framework
State.InitGoblin(graphics, Content, "");

// Initialize the scene graph
scene = new Scene();
```

```
// Set the background color to CornflowerBlue color.
// GraphicsDevice.Clear(...) is called by
//Scene object with this color.
scene.BackgroundColor = Color.CornflowerBlue;

// Custom method for creating a 3D object
CreateObject();

// Set up the lights used in the scene
CreateLights();

// Set up the camera, which defines the eye location
//and viewing frustum
CreateCamera();

// Show Frames-Per-Second on the screen for debugging
State.ShowFPS = true;
```

Following this are three private methods we created to better organize the creation of important scene elements: `CreateObject()`, `CreateLights()`, and `CreateCamera()`.

GRAPHICAL OBJECTS

Our first consideration is our graphical objects, which we will specify in our `CreateObject()` method. (Note that this is just one possible way of structuring a program. Objects could instead be specified in the `Initialize()` method or elsewhere.)

We will first specify the geometric shape and geometric transformations associated with each object.

SHAPE

The shape of an object can be defined by one of the Goblin XNA pre-built simple geometric models, by a model file created in a 3D modeling application (e.g., an *fbx* file), or by creating geometry "from scratch." In our example, we will be using a simple geometric model of a sphere defined by Goblin XNA.

In initializing the `Sphere` object, the first argument specifies the radius of the sphere. The second argument specifies the number of "slices" around the upright *y*-axis (think of the slices as being defined by "lines of longitude" around the sphere). The third argument specifies the number of "stacks" perpendicular to the *y*-axis (think of the stacks as being defined by "lines of latitude" around the sphere). These two arguments define the resolution of the sphere, which is actually a convex polyhedral approximation of a sphere; the more slices and stacks that are specified, the closer the

resulting shape approximates a true sphere (and the more polygons are used in that approximation).

TRANSFORMATIONS

A Goblin XNA `Sphere` is defined to be centered about the origin of the coordinate system. If we wish to change the location of a shape such as a `Sphere`, or, for that matter, its scale or its orientation, we can do that with geometric transformations that systematically modify the values of the coordinates with which the shapes are defined. For example, a scale transformation (change in size) can be specified as a vector of three floats, one for each of the *x*, *y*, and *z* axes. Similarly, a translation (change in position) can also be specified as a vector of three floats, specifying displacements parallel to the *x*, *y*, and *z* axes. In the code below, the sphere is translated zero units along the *x*-axis and *y*-axis and –5 units along the *z*-axis. Note that XNA Game Studio uses a right-handed coordinate system; that is, the *x*-axis increases positively to the right, the *y*-axis increases positively upwards, and the *z*-axis increases out of the screen, towards the viewer.

There is a set of programs (with source code) that can help you gain an understanding of a variety of aspects of graphics programming at http://www.xmission.com/~nate/tutors.html. They allow you to control interactively the parameters that determine what you see, including the parameters to scale, rotation, and translation transformations. While these programs are written for OpenGL, there is a document at http://graphics.cs.columbia.edu/courses/csw4172/usingNateRobinsTutorsWithGoblinXNA.pdf that provides a mapping between the relevant parts of the OpenGL API and the Goblin XNA and XNA Game Studio APIs.

If you feel you need to brush up on your vector math, you can find a remedial tutorial at http://chortle.ccsu.edu/VectorLessons/vectorIndex.html.

CREATING THE OBJECTS

Each object must be inserted in the right place in the scene graph. In this example, we add the sphere as a child of the sphere transformation node, so that its transformations affect the sphere. The sphere transformation is then added to the root node of the scene as a child. Here is the `CreateObject()` method:

```
private void CreateObject() {

      // Create a transform node to define the
      // transformation of this sphere
      // (Transformation includes translation, rotation,
      // and scaling)
```

```
            TransformNode sphereTransNode = new TransformNode();

            // We want to scale the sphere by half in all three
            // dimensions and translate the sphere 5 units back
            // along the Z axis
            sphereTransNode.Scale = new Vector3(0.5f, 0.5f,
                  0.5f);
            sphereTransNode.Translation = new Vector3(0, 0, -5);
            // Create a geometry node with a model of a sphere
            // NOTE: We strongly recommend you give each geometry
            // node a unique name for use with
            // the physics engine and networking; if you leave
            // out the name, it will be automatically generated

            GeometryNode sphereNode = new GeometryNode("Sphere");
            sphereNode.Model = new Sphere(3, 60, 60);

            // Create a material to apply to the sphere model
            Material sphereMaterial = new Material();
            sphereMaterial.Diffuse = new Vector4(0.5f, 0, 0, 1);
            sphereMaterial.Specular = Color.White.ToVector4();
            sphereMaterial.SpecularPower = 10;

            // Apply this material to the sphere model
            sphereNode.Material = sphereMaterial;

            // Child nodes are affected by parent nodes. In this
            // case, we want to make
            // the sphere node have the transformation, so we add
            // the transform node to
            // the root node, and then add the sphere node to the
            // transform node.
            scene.RootNode.AddChild(sphereTransNode);
            sphereTransNode.AddChild(sphereNode);

      }
```

LIGHTING

Next we set up the lighting for the scene:

```
      private void CreateLights()
      {
            // Create a directional light source
            LightSource lightSource = new LightSource();
            lightSource.Direction = new Vector3(-1, -1, -1);
            lightSource.Diffuse = Color.White.ToVector4();
            lightSource.Specular = new Vector4(0.6f, 0.6f,
                  0.6f, 1);

            // Create a light node to hold the light source
```

```
        LightNode lightNode = new LightNode();
        lightNode.LightSource = lightSource;

        // Add this light node to the root node
        scene.RootNode.AddChild(lightNode);
    }
```

What we are doing in the above assignments is creating a new `LightSource` object and then specifying its properties. The `LightNode` object allows us to hang our light source on the tree of nodes that makes up the scene graph. If a `LightNode` is global (the default), then the `LightSource` added to that node can potentially affect every object in the scene graph; if a `LightNode` is local, then it can potentially affect only its sibling nodes and their descendants.

CAMERA

A camera provides a first-person view of a scene, *projecting* the 3D scene objects onto a 2D planar *view plane*. Since it is part of the scene graph, transformations applied to its parent node will affect it, too.

The camera specifies a *view volume* that, for a perspective projection, is a truncated pyramid (*frustum*) whose apex (center of projection) is at the origin, and whose center line extends along the –*z*-axis. The shape of the pyramid is conventionally defined in terms of a vertical *field of view* specified in radians along the *y*-axis, an *aspect ratio* (of width/height), and *near* and *far clipping planes* that bound the view volume, preventing objects from being seen that are too close to or too far from the center of projection. Setting the clipping planes carefully can reduce visual overload and speed up rendering, since objects outside of the view volume that they bound will not be rendered, and can often be efficiently eliminated from much of the graphical processing needed for objects within the frustum.

```
private void CreateCamera()
{
        // Create a camera
        Camera camera = new Camera();
        // Put the camera at the origin
        camera.Translation = new Vector3(0, 0, 0);
        // Set the vertical field of view
        // to be 60 degrees
        camera.FieldOfViewY = MathHelper.ToRadians(60);
        // Set the near clipping plane to be
        // 0.1f unit away from the camera
        camera.ZNearPlane = 0.1f;
        // Set the far clipping plane to be
        // 1000 units away from the camera
        camera.ZFarPlane = 1000;
```

```
            // Now assign this camera to a camera node,
            // and add this camera node to our scene graph
            CameraNode cameraNode = new CameraNode(camera);
            scene.RootNode.AddChild(cameraNode);

            // Assign the camera node to be our
            // scene graph's current camera node
            scene.CameraNode = cameraNode;
      }
```

UPDATE AND DRAW

These two inherited methods allow the application's logic to be executed and its state to be updated. `Update()` is often used to make abstract changes to the application's state and to do bookkeeping for the application, whereas `Draw()` is reserved for changes that directly affect the on-screen appearance of the application. Normally these two methods are called one after the other in a continuous loop, but it is possible to decouple the two so that updates take place more often than screen redraws:

```
      /// Allows the game to run logic such as updating the
      /// world, checking for collisions, gathering input,
      /// and playing audio.

      protected override void Update(GameTime gameTime) {
            scene.Update(gameTime.ElapsedGameTime,
                  gameTime.IsRunningSlowly, this.IsActive);
      }
```

In the `Draw()` method, below, we write "Hello, World" in the middle of the screen:

```
      /// This is called when the game should draw itself.
      protected override void Draw(GameTime gameTime) {

            // Draw a 2D text string at the center of the
            // screen.

            UI2DRenderer.WriteText(
                  Vector2.Zero, "Hello World!!",
                  Color.GreenYellow, textFont,
                  GoblinEnums.HorizontalAlignment.Center,
                  GoblinEnums.VerticalAlignment.Center);

            // Draw the scene graph
            scene.Draw(gameTime.ElapsedGameTime,
                  gameTime.IsRunningSlowly);
      }
```

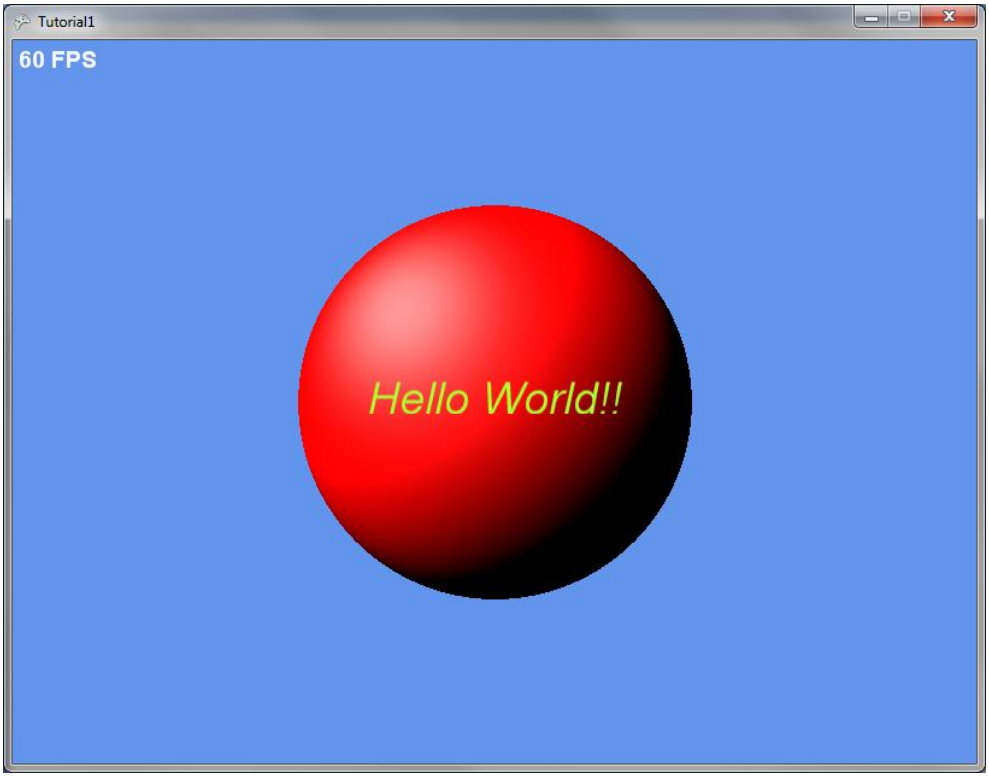
DISPOSE

Before the application terminates, you need to release all of the resources allocated in the scene graph by calling its `Dispose()` method. Make sure to override the `Game.Dispose(..)` function and call `Scene.Dispose()` in this overridden method.

```
 /// This is called when the game terminates.
protected override void Dispose(bool disposing) {
     scene.Dispose()
}
```

RUNNING THE APPLICATION

If there are no errors, you can run the application by choosing the "Debug" menu and then either "Start Debugging" or "Start without Debugging". Alternatively, you can hit the F5 key for a debugging run or Ctrl-F5 for a live run. Or, you can simply choose "Debug" or "Release" next to the green triangular "play" icon, and then click that icon. If all goes well, you will see this:



SCENE GRAPH

The design of the Goblin XNA scene graph is similar to that of OpenSG. Our scene graph currently consists of ten node types:

- Geometry
- Transform
- Light
- Camera
- Particle
- Marker
- Sound
- Switch
- LOD (Level Of Detail)
- Tracker

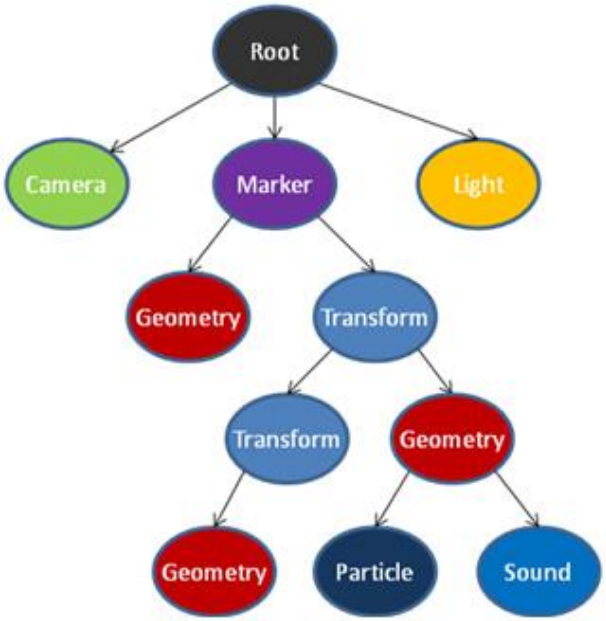


Figure 4: Goblin XNA scene graph hierarchy

An example scene graph hierarchy is illustrated in Figure 4. The scene graph is rendered using preorder tree traversal.

Only the following six node types can have child nodes: Geometry, Transform, Marker, Switch, LOD, and Tracker. (All child-bearing nodes inherit from `BranchNode`.) Each of these nodes has a `Prune` property (false by default), which can be set to true to disable the traversal of its children. There is a separate (and different) `Enabled` property (true by default) that all nodes have, which, when set to false, not only disables the traversal of a node's children, but also disables traversal of the node itself. Detailed descriptions of all node types are provided in the following section. In addition to the specific information associated with each node type by Goblin XNA, you can also associate arbitrary user-defined information with a node by setting the `UserData` property.

NODE TYPES

GEOMETRY

A Geometry node contains a geometric model to be rendered in the scene. For example, a Geometry node might represent a 3D airplane model in a flight simulator game. A Geometry node can have only one 3D model associated with it. If you want to render multiple 3D models, then you will need to create multiple Geometry nodes. A geometric model can be created by either loading data from an existing model file, such as an *.x* or *.fbx* file (see Tutorial 2), or using a list of custom vertices and indices

(see Tutorial 7). Goblin XNA provides several simple shapes, such as Cube/Box, Sphere, Cylinder/Cone, ChamferCylinder, Capsule, Torus, and Disk, similar to those in the OpenGL GLUT library. `Billboard` and `Text3D` geometries are also supported as of v4.0. In addition to the geometric model itself, a Geometry node also contains other properties associated with the model. Some of the important properties include:

- Material properties, such as color (diffuse, ambient, emissive, and specular color), shininess, and texture, which are used for rendering.
- Physical properties, such as shape, mass, and moment of inertia, which are used for physical simulation. (See the section on Physics for details on the supported physics engines.)
- Occlusion property, which determines whether the geometry will be used as an occluder that occludes the virtual scene, typically used in augmented reality applications. When the geometry is set to be an occluder, the virtual geometry itself will not be visible, but it will participate in visible-surface determination, blocking the view of any virtual objects that are behind it relative to the camera. Occluder geometry is typically transparent because it corresponds to real objects (visible in the view of the real world in an augmented reality application) with which the virtual object interacts.
- IgnoreDepth property, which determines whether the geometry should be rendered in front of all other geometries even if there are geometries closer to the eye than this geometry by ignoring the depth information. If there are more than two geometries with `IgnoreDepth` set to true, then the rendering order depends on the order the node is traversed in the scene graph.

Geometry nodes are transformed by physical simulation if `AddToPhysics` is set to true, as described in the section on Physics. Since a Geometry node can have children, those children will also be affected by any transformation that affects a Geometry node. Consider the following example: if you are not using physical simulation, then the two hierarchies shown in Figure 5 accomplish the same thing: both the Sound and Geometry nodes will be influenced by the Transform node.

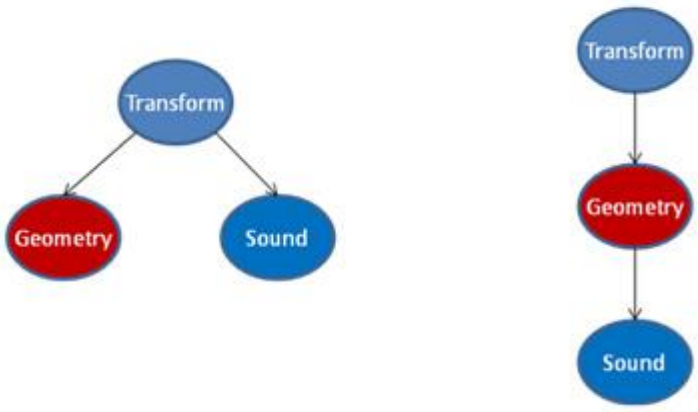


Figure 5: (a) Sound is not affected by Geometry node transformation. (b) Sound is affected by Geometry node transformation.

However, if physical simulation is being performed (see the section on Physics), then the Sound node

will be transformed differently in the two hierarchies, since the Geometry node's transformation is not only affected by the Transform node, but also by the physical simulation. For example, in Figure 5(a), if the Sound node is used to play a 3D sound, then you will hear the sound at the location specified by the Transform node, which is not affected by the physical simulation. In contrast, in Figure 5(b) you will hear the sound at the location of the 3D model associated with the Geometry node, because it is affected by the physical simulation.

As of v3.5, the Network property of the Geometry node has been moved to the SynchronizedGeometry node, which extends the Geometry node. This makes it possible for the programmer to determine which Geometry nodes should be synchronized among multiple networked machines and which should be local.

TRANSFORM

A Transform node modifies the transformation (e.g., translation, rotation, and scale) of any object beneath it in the hierarchy that contains spatial information. For example, if you add a Camera node to a Transform node, and modify the transformation of the Transform node, the spatial information of the Camera node will also change.

There are two ways to set the transformation of a Transform node. One way is to set each transformation component (pre-translation, scale, rotation, and (post-) translation) separately, causing the composite transformation matrix to be computed automatically from these values. No matter the order in which the components of the transformation are set, they will be applied such that any child of the Transform node will first be translated by the pre-translation component, then scaled by the scale component, next rotated by the rotation component, and finally translated by the (post-)translation component. An alternative is to directly set the transformation matrix. Note that the last approach used determines the transformation. For example, if you set the pre-translation, scale, rotation, and (post-)translation separately, and then later on assign a new transformation directly, the node's transformation will be the newly assigned one. If you want to switch back to using a transformation determined by the pre-translation, scale, rotation, and (post-)translation, you will need to assign a value to one of these components, causing the node's transformation to once again be composed from the pre-translation, scale, rotation, and (post-)translation.

Note: Prior to v3.4, scale and rotation were performed in the wrong order. (This would have caused incorrect behavior *only* if the *x*, *y*, and *z* scale factors were not identical.) Starting in v3.4, the `PreTranslation` property was added, and `PostTranslation` was added as an alias for `Translation`.

LIGHT

A Light node contains "light sources" that illuminate the 3D models. Light source properties differ based on the type of light, except for the diffuse and specular colors, which apply to all types. There are three types of simplified lights that have typically been used in "fixed pipeline" real-time computer graphics: directional, point, and spot lights.

A directional light has a direction, but does not have a position. The position of the light is assumed to be infinitely distant, so that no matter where the 3D model is placed in the scene, it will be illuminated from a constant direction. For example, the sun, as seen from the earth on a bright day, is often conventionally modeled as a directional light source, since it is so far away.

A point light has a position from which light radiates spherically. The intensity of a point light source attenuates with increased distance from the position, based on attenuation coefficients. For example, a small bare light bulb is often conveniently modeled as a point light.

A spot light is a light that has a position and direction, and a cone-shaped frustum. Only the 3D models that fall within this cone shaped frustum are illuminated by a spot light. As its name implies, a spot light can be used to model a simplified theatrical light.

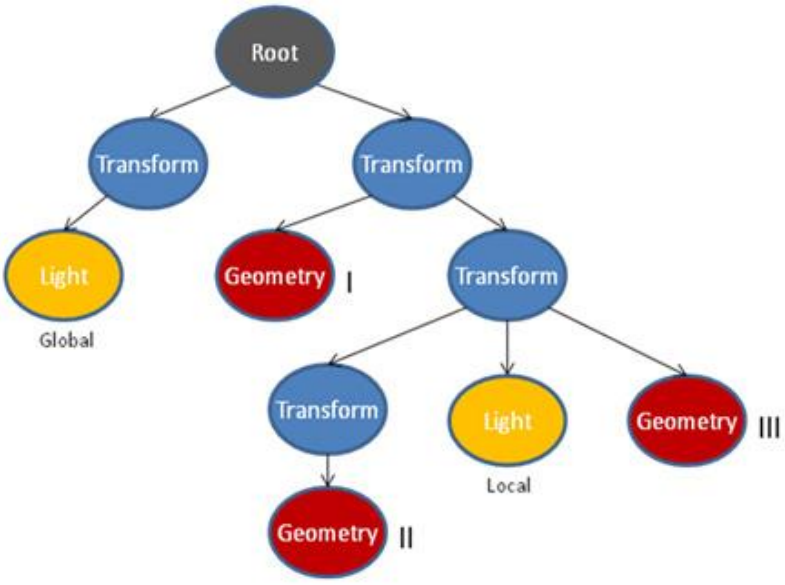


Figure 6: Global and local Light nodes

A Light node can contain one light source and an ambient light color, and the entire node can either be global or local. A global Light node illuminates the entire scene in the scene graph. In contrast, a local light node illuminates only a part of the scene: the Light node's sibling nodes and all their descendant nodes. For example, in Figure 6, the global Light node (marked "Global") illuminates all Geometry nodes in the scene (Geometry nodes I, II, and III), while the local Light node (marked "Local") illuminates only Geometry nodes II, and III (i.e., the siblings of the local Light node and their descendants). Furthermore,

note that if the global Light node in Figure 6 were a local Light node, then it would illuminate none of the Geometry nodes in the scene graph, because this Light node has no siblings.

CAMERA

A Camera node defines the position and visible frustum of the viewer (i.e., the *view volume* containing what you see on your display). The visible frustum of the viewer is defined by the vertical field of view, aspect ratio (ratio of frustum width to height), near clipping plane, and far clipping plane, as shown in Figure 7. The initial view direction is toward the –z direction with an up vector of +*y*. The Camera node rotation property modifies this view direction by applying the given rotation to the initial view direction. You can create a view frustum that is a regular pyramid by assigning values

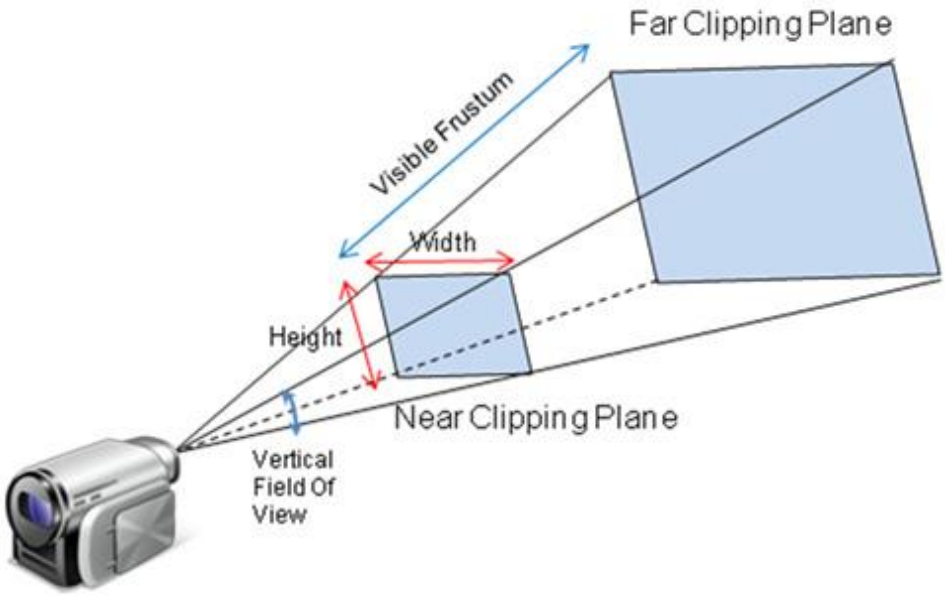


Figure 7: Camera geometry

to these parameters, causing the view and projection matrices to be computed automatically. Alternatively (e.g., if you want to create a view frustum that is not a regular pyramid), you can assign values directly to the view and projection matrices. See the description of the `Scene` class to learn how to render the view from a camera.

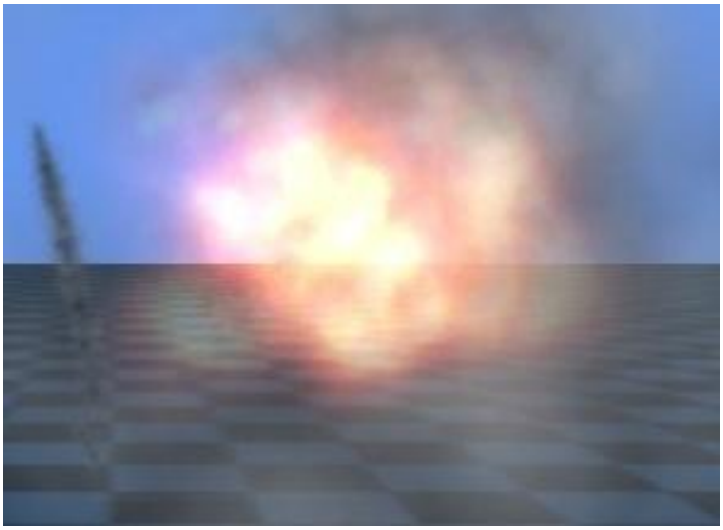


Figure 8: Explosion particle effect (courtesy of the XNA Creators Club)

PARTICLE

A Particle node contains one or more particle effects, such as fire, smoke, explosions (Figure 8), and splashing water. A particle effect has properties such as texture, duration before a particle disappears, start size, end size, and horizontal and vertical velocities. Goblin XNA provides a small set of particle effects, including fire and smoke. You can also modify the properties of an existing particle effect to create your own particle effect. Please see Tutorial 6 to learn about using simple particle effects. (Note: to speed up rendering, make sure to change the following properties of the content processor of the textures used for particle effects: Set `GenerateMipmaps` to `true`, and `TextureFormat` to `DxtCompressed`.)

MARKER

A Marker node modifies the transformations of its descendant nodes, similar to a Transform node. However, the transformation is modified based on the 6DOF (six-degrees-of-freedom) pose matrix computed for an array of one or more fiducial markers. **Fiducial markers** are geometric patterns, typically printed out on paper or cardboard, that are viewed by a video camera. The video is processed interactively by computer vision algorithms that can find the image of each clearly visible marker and compute the position and orientation of that marker relative to the camera to determine its pose matrix. Before you can use a Marker node, you will need to initialize the marker tracker and video capture devices, as explained in those sections. You can then create a Marker node to track a single marker or a specific marker array.

Goblin XNA currently supports the ALVAR marker tracker.

You can pass either one or two parameters. If you want to track a single marker, then you need to pass the ID of the marker as an integer in the first parameter of `markerConfigs` with an optional marker size as a double in the second parameter. If you want to track an array of markers, then you need to pass a file name that contains the marker configuration file (see below) in the first parameter of

`markerConfigs`, and an array of integer marker IDs used in the marker array in the second parameter.

NOTE: The marker configuration file can be automatically generated by the MarkerLayout program provided in the tools directory.

A Marker node not only provides a computed pose matrix, but also supports smoothing out the sequence of pose matrices detected over time if you set the `Smoother` property. You can either use the double exponential smoothing implementation (`DESSmoother`) supported by Goblin XNA or create your own smoothing filter by implementing the `ISmoother` interface.

The `DESSmoother` smoothing alpha determines how smoothing is performed. The smoothing alpha ranges between 0 and 1, excluding 0. The larger the value, the more the smoothed pose matrix is biased toward the more recently tracked poses. (A value of 0 would indicate that the recently tracked pose matrix would not be taken into consideration, and is therefore not allowed.) A value of 1 indicates that the previously tracked pose matrix will not be taken into consideration, so that only the most recently tracked pose matrix will be used. For a fiducial marker array that is expected to move quickly, we recommend that you use a higher smoothing alpha, since the result matrix after smoothing will weight recently tracked pose matrices more heavily than older ones; for a fiducial marker array that is expected to move slowly, we recommend that you use a lower smoothing alpha.

To track a larger area with multiple marker arrays that can be rearranged on the fly, use the MarkerBundle node, which extends the Marker node. The MarkerBundle node consists of one base marker array and multiple supporting marker arrays. The base marker array provides the base transform of the node, and the supporting marker arrays are used to compute the relative transform from the base marker array.

SOUND

A Sound node contains information about a 3D sound source that has a position, velocity, and forward and up vector. Before you can use a Sound node to play audio, you will first need to initialize the XNA audio engine. Goblin XNA provides a wrapper for the XNA audio engine, and the section on Sound describes how you can initialize the sound wrapper class. Once the audio engine is initialized, you can call the `SoundNode.Play(String cueName)` function to play 3D audio (Note: To produce spatial sound effects, such as the Doppler effect or distance attenuation, you will need to set these effects properly when you create the XACT project file. Please see the section on Sound for details.)

Alternatively, you can play 3D audio by using the `Sound.Play3D(String cueName, IAudioEmitter emitter)` function directly. However, it is much simpler to attach a Sound node to the scene graph, so that you do not need to worry about updating the position, velocity, forward and up vector of the 3D sound source. For example, if you attach a Sound node to a Geometry node, then the 3D sound follows wherever the Geometry node moves. Otherwise, you will have to create a class that implements the `IAudioEmitter` interface, update the position, velocity, forward and up vector yourself, and then pass the 3D audio cue name and your class that implements `IAudioEmitter` to `Sound.Play3D(String cueName, IAudioEmitter emitter)`.

SWITCH

A Switch node is used to select a single one of its child nodes to traverse. The `SwitchID` property denotes the index in the Switch node's children array (starting from `0`), not the actual ID of a child node. For example, if you add a Geometry node, a Transform node, and a Particle node to a Switch node, in that order, and set `SwitchID` to `1`, then only the Transform node will be traversed during scene graph traversal. The default `SwitchID` is `0`, so the first child node added to a Switch node is traversed by default.

LOD

An LOD (Level of Detail) node is used to select a single model to be rendered from a list of models, each of which is assumed to have a different level of detail. The LOD node extends the Geometry node; however, instead of a single `IModel` object, it requires a list of `IModel` objects. The `LevelOfDetail` property is used to select which model to render, assuming the first model in the list has the finest level of detail and the last model has the coarsest level of detail. Instead of manually modifying the `LevelOfDetail` property, you can also make the node automatically compute the appropriate level of detail to use based on the distances set in the `AutoComputeDistances` property by setting `AutoComputeLevelOfDetail` to `true`. Please see the API documentation for a description of how you should set these distances.

TRACKER

A Tracker node is quite similar to a Marker node. The only parameter you need to set is the device identifier that specifies which 6DOF tracker to use. Once you set the device identifier, the world transformation is automatically updated, and any nodes added below this Tracker node are affected, as well. You can set the `Smoother`

property if you want to apply a smoothing filter to the transformation of the 6DOF device, as with the Marker node, described above.

SCENE

`Scene` is the Goblin XNA scene graph class. Once you understand how each node type is used and how it affects other node types, you can start creating your scene graph tree by adding nodes to `Scene.RootNode`. (`Scene` is not a static class, so you first need to instantiate the class properly.) Since `Scene.RootNode` is settable (and must be non-null), if you have multiple different scene graph structures and want to switch between them as an atomic action, then you can simply assign the root of any of your scene graph structures to `Scene.RootNode`, as desired.

You should have at least one Camera node in your scene graph; otherwise, you will not see anything on your screen. Once you create a Camera node, you can add it to the scene graph. Since the scene graph can contain multiple Camera nodes, it is necessary to specify which Camera node is the active one. This is done by assigning it to `Scene.CameraNode`. However, this does not mean that you are restricted to a single viewport. You can have multiple viewports, with each viewport assigned to a different camera, as in a two-player game in which each player has a viewport that is half of the entire screen, with each viewport showing its player's view. (As described below, Tutorial 13 demonstrates how you can render two viewports with different cameras for stereo display to a single user on Vuzix Wrap 920 eyewear.)

As of version 4.0, you need to call `Scene.Update(...)`, `Scene.Draw(...)`, and `Scene.Dispose()` in appropriate functions (as mentioned here). This change was necessary in order to make it possible to integrate Goblin XNA with WPF and Silverlight.

STEREOSCOPIC GRAPHICS

We support stereoscopy through the `StereoCamera` class, assuming that an appropriate stereo device is present. Currently, the only stereo devices supported are the Vuzix iWear VR920 and Wrap 920 head-worn displays. To render in stereo, you will need to use the `StereoCamera` class instead of the `Camera` class when you create a `CameraNode`. `StereoCamera` is a subclass of `Camera`: When you are rendering graphics that is *not* overlaid on real camera views of the physical world, the only additional property you will need to set is `InterpupillaryDistance` (IPD), which defines the distance between the user's left and right eyes (cameras). If the center of the stereo display is not in the middle of the two eyes, you can use `InterpupillaryShift` property to shift the center point. You can also modify the focal length from the eye by modifying the `FocalLength` property. For Augmented Reality, in which graphics is rendered over real camera views, you should not set these

properties. Instead, you should use our stereo calibration tools to calibrate your physical cameras' intrinsic parameters, stereo separation, and field of view adjustments. Then, you should load the generated calibration file as shown in Tutorial 13. For details, please see the section on Stereoscopic Augmented Reality.

Tutorial 13 shows how to set up stereoscopic rendering correctly for the Vuzix iWear VR920 and Wrap 920. Once the camera is set to stereo, the `Scene` class renders the scene for the left and right eyes, in turn, if the Vuzix iWear VR920 is used. To display the left and right eye scenes correctly to the corresponding eyes using the Vuzix iWear VR920, you will need to synchronize the device after rendering each scene. As demonstrated in Tutorial 13, you will need to set up the stereo rendering yourself for the Vuzix Wrap 920.

Note that a stereo application should be set to run in full-screen mode for display on the Vuzix iWear VR920. Otherwise, the synchronization will not work properly and the left and right eye views will not be assigned correctly to the left and right displays.

6DOF AND 3DOF TRACKING AND INPUT DEVICE ABSTRACTION

We currently support 3DOF (three–degrees-of-freedom) orientation tracking and 6DOF (six–degrees-of-freedom) position and orientation tracking using the 6DOF ALVAR optical marker tracking systems, 3DOF and 6DOF InterSense hybrid trackers, and the 3DOF Vuzix iWear VR920 and Wrap Tracker 6TC (currently for 3DOF only) trackers. The ALVAR interface is provided though the Marker node, as described in the Marker section, as well as the `Scene` class. In contrast, the InterSense trackers, GPS, and Vuzix orientation trackers (iWearTracker and Wrap Tracker 6TC) should be interfaced through the `InputMapper` class, which provides hardware device abstractions for various input devices. The other input devices currently supported through the `InputMapper` class are mouse, keyboard, and a combination of mouse and keyboard that simulates a 6DOF input device. All of these device classes are singleton classes, so you need to use their `Instance` properties to access them.

6DOF INPUT DEVICES

A 6DOF input device provides position (*x*, *y*, *z*) and orientation (yaw, pitch, roll) information. Any 6DOF input device implements the `InputDevice_6DOF` interface. The position and orientation can be accessed through the `InputMapper.GetWorldTransformation(String identifier)` method if the device is available. For the specific value of identifier, please refer to the `Identifier` property of each device classes.

INTERSENSE HYBRID TRACKERS

To use an InterSense hybrid tracker, you will need to get an instance of the `InterSense` class under the `GoblinXNA.Device.InterSense` package. Then add this instance to the `InputMapper` class using the `InputMapper.Add6DOFInputDevice(…)` method and reenumerate the input devices by calling `InputMapper.Reenumerate()`. An InterSense hybrid tracker can be connected either through a direct serial port connection or a network server connection. If you want to connect the tracker through a network server, then you will need to pass the appropriate host name and port number to the `Initialize(…)` function. Otherwise, simply call `Initialize()` to connect through a direct serial port connection.

The InterSense support software can handle up to eight stations, and you can access each station by modifying the `CurrentStationID` property and then passing the appropriate identifier when you call the `InputMapper.GetWorldTransformation(String identifier)` function. If the station is not available, you will get an identity matrix. If an orientation

tracker is used (instead of one that also senses position), then the translation components of the transformation will always be zeros.

VUZIX IWEAR VR920 AND WRAP TRACKER 6TC ORIENTATION TRACKER

Once you install the appropriate Vuzix SDK, you can use the `iWearTracker` class for 3DOF orientation tracking. Similar to the Goblin XNA support for InterSense trackers, you need to get an instance of `iWearTracker` and initialize it. Then, add this instance to the `InputMapper` class using the `InputMapper.Add6DOFInputDevice(…)` method and reenumerate the input devices by calling `InputMapper.Reenumerate()`. You can obtain orientation information either individually by accessing the `Yaw`, `Pitch`, or `Roll` properties or in combination by accessing the `Rotation` property.

SIMULATED 6DOF INPUT USING THE MOUSE (GENERICINPUT)

The `GenericInput` class makes it possible for a mouse to support a simulated 6DOF input device in which mouse dragging and wheeling controls both orientation and translation relative to the `GenericInput.BaseTransformation`. This can be used for transforming objects and cameras.

To modify orientation, hold down the right mouse button and drag the mouse around the screen to rotate around the *x*-axis and *y*-axis of the `GenericInput.BaseTransformation`.

To modify translation, hold down the left mouse button and drag the mouse parallel to the *x*-axis (of the screen coordinate system) for left and right translation, and parallel to the *y*-axis (of the screen coordinate system) for forward and backward translation. To modify up and down translation, hold the middle mouse button and drag the mouse parallel to the *y*-axis (of the screen coordinate system).

An additional interaction is provided for use when a camera is being transformed, and is based on rolling the mouse wheel forward or backward. This causes translation by a scaled copy of the vector from the center of the near clipping plane to the projection of the mouse cursor onto the far clipping plane.

The scale factor used for translation and rotation can be changed. If the object you want to control has position or rotation other than `Vector3.Zero` or `Orientation.Identity`, then you should set the `InitialTranslation` and/or `InitialRotation` property to be the position and/or orientation of the object.

NON-6DOF INPUT DEVICE

Any input devices that do not support 6DOF input are in this category (e.g., mouse, keyboard, gamepad, and GPS). Unlike 6DOF input devices, these devices do not provide the same type of input, so there is no unified method like `GetWorldTransformation(String identifier)`. Thus, you will need to access individual input device classes, instead of accessing them through the `InputMapper` class. However, all of the non-6DOF input device classes provide `TriggerDelegates(…)` methods that can be used to programmatically trigger some of the callback functions defined in each individual input device class. For example, even if the actual keyboard key is not pressed, you can use this method in your program to trigger a key press event.

MOUSE

The `MouseInput` class supports delegate/callback-based event handling for mouse events (e.g., mouse press, release, click, drag, move, and wheel move). The `MouseInput` class has public event fields, such as `MouseInput.MousePressEvent`, and a `HandleMousePress` delegate can be added to this event field, just like any other C# event variables. Please see Tutorial 4 to learn about how to add a mouse event delegate.

KEYBOARD

The `KeyboardInput` class supports delegate/callback-based event handling for keyboard events (e.g., key press, release, and type). Like the `MouseInput` class, the `KeyboardInput` class also has public event fields, such as `KeyboardInput.KeyPressEvent`. Please see Tutorial 2 to learn about how to add a keyboard event delegate.

GPS (GLOBAL POSITIONING SYSTEM)

The GPS class supports reading GPS receiver data from a serial port (NMEA, GPRMC, and GPGGA formats are supported) and parses the data into useful coordinates (e.g., latitude, longitude, and altitude). The GPS device is assumed to be connected on a serial port. The class has been tested on GPS devices connected through USB and Bluetooth. To use the class, create a new instance of `GPS` and add a `GPSListener` delegate using the `AddListener(GPSListener listener)` method or poll the individual properties (e.g., `Latitude`).

AUGMENTED REALITY

Figure 9: Virtual dominoes are overlaid on a real video image using Goblin. XNA.

One of the most important goals of the Goblin XNA framework is to simplify the creation of augmented reality applications. Augmented reality (AR) [Azuma 96, Feiner 02] refers to augmenting the user's view of the real world with a virtual world interactively, such that the real and virtual worlds are geometrically registered with each other. In many AR applications, the real world is often viewed through a head-worn display or a hand-held display. In so-called "video see-through" applications, an image of the real world obtained by a camera is augmented with rendered graphics, as shown in Figure 9: Virtual dominoes are overlaid on a real video image using Goblin. XNA. Two important issues that Goblin XNA addresses for video see-through AR are combining rendered graphics with the real world image and 6DOF pose (position and orientation) tracking for registration. To support video devices for capturing the real image, we support three different types of video-capture libraries: DirectShow, OpenCV, and PGRFly (from Point Grey Research, and used only with Point Grey cameras).

CAPTURING VIDEO IMAGES

To capture live video images, you will need to instantiate one of the classes that implements the `IVideoCapture` interface (`GoblinXNA.Device.Capture.IVideoCapture`). Goblin XNA currently provides four different implementations of the `IVideoCapture` interface: `DirectShowCapture` (and the alternative `DirectShowCapture2`), `OpenCVCapture`, `PointGreyCapture`, and `NullCapture`.

If you are using a standard web camera, then `DirectShowCapture` is suitable.

If you have too many DirectShow filters installed on your computer (e.g., DirectShow filters are automatically installed by software such as Nero or Adobe Premiere) , the `DirectShowCapture` implementation may not work, for reasons we have not yet

determined. In those cases, you will need to use the `DirectShowCapture2` implementation instead, which is slower than the `DirectShowCapture` implementation because of an additional class layer.

If you are using a Point Grey camera (e.g., a Firefly or Dragonfly), then `PointGreyCapture` is suitable.

If you plan to use OpenCV with Goblin XNA (Tutorial 15), then it's best to use `OpenCVCapture`. OpenCV caps the rendering speed to 30 FPS, which is the same speed as the video frame acquisition.

`NullCapture` is used to render a static image from an image file (e.g., a JPEG file) for testing purposes.

If you want to use another video decoding library, you will need to create your own class that implements the `IVideoCapture` interface. After instantiating one of the `IVideoCapture` implementations, you will need to initialize it by calling `IVideoCapture.InitVideoCapture(…)` with appropriate parameters, as demonstrated in Tutorial 8. Once it is initialized, you are ready to start capturing video images.

There are three different ways to use the Goblin XNA video capture device interface, depending on your needs. First, if you want to use the captured video as the scene background, and also pass it to the optical marker tracker for pose tracking, you will need to add the initialized `IVideoCapture` implementation to the Scene class by calling `Scene.AddVideoCaptureDevice(…)`. In this case, make sure to set `Scene.ShowCameraImage` to `true`, so that the video will be displayed in the background.

Second, if you also want to use the captured video for additional processing, such as face or gesture recognition, you can access the specific method for each of the `IVideoCapture` implementations that retrieves the video image. For example, once the capture device is added to the Scene and `Scene.ShowCameraImage` is set to true, you can set the `IVideoCapture.CaptureCallback` property, which is a callback function invoked whenever a new video image is grabbed. The format of the `imagePtr` parameter is defined by the `ImageFormat` parameter passed in the `InitVideoCapture(…)` function. The integer array contains ARGB values of each pixel in each integer value (8 bit for each channel, and the alpha channel is ignored). Note that if `Scene.MarkerTracker` is `null`, then the passed `imagePtr` parameter will be `IntPtr.Zero`, and if `Scene.ShowCameraImage` is false, then the passed integer array is `null`, for optimization purposes. If you want to force those passed parameters to be valid, you need to set either/both `Scene.NeedsImagePtr` or/and `Scene.NeedsImageData` to `true`.

Third, if you simply want to acquire the captured image and neither want to use it for optical marker tracking nor show the video image on the background, then do not add the `IVideoCapture` implementation to the Scene class. Instead, simply call `GetTextureImage(…)` whenever you need it.

If you do not have a physical web camera, or want to capture video streamed from a file stored in your local drive (e.g., for overlaying an existing video for regression testing), then you can use an application such as VCam™ from e2esoft to simulate a virtual web camera. For example, after capturing a video file containing optical markers, you can repeatedly use it in developing and testing your program.

OPTICAL MARKER TRACKING

After at least one video capture device is added to the `Scene` class after initialization, you should instantiate one of the `IMarkerTracker` (`GoblinXNA.Device.Vision.Marker.IMarkerTracker`) implementations. Goblin XNA includes marker tracker classes using the ALVAR library. After you instantiate one of the `IMarkerTracker` implementations (e.g., `ALVARMarkerTracker`, if you are using ALVAR), you will need to initialize the tracker by calling `InitTracker(…)` with appropriate parameters.

For `ALVARMarkerTracker`, you can pass either four or six parameters to the `InitTracker(…)` method. The first and second parameters are the width and height of the image to be processed by ALVAR. If you're using one of the `IVideoCapture` implementations, you can simply pass the `IVideoCapture.Width` and `IVideoCapture.Height` properties. The third parameter is the camera calibration file. The ALVAR package provides a sample project (*SampleCamCalib*) that automatically calibrates your camera and outputs a calibration file. (To calibrate your camera, you will need to use the checkerboard image provided in the Alvar.ppt file under its *doc* directory.) Alternatively, you can use the *CameraCalibration* program under *GoblinXNAv4.0\tools*, which wraps the ALVAR calibration program in C#. There is more detailed documentation at the Goblin XNA codeplex site. The fourth parameter is the size of the marker (in terms of the units used in the application). The fifth and sixth parameters are optional, and they are the marker resolution and the margin width, respectively.

Once you initialize the marker tracker, you will need to assign the marker tracker to the `Scene.MarkerTracker` property. Then, you are ready to create Marker nodes that can track a specific marker array defined in the configuration file. Please refer to the Marker section to learn how to create and use a Marker node.

If you want to use a static image to perform the tracking instead of live video images, then you will need to use the `NullCapture` class instead of the

`DirectShowCapture`, `OpenCVCapture`, or `PointGreyCapture` implementations.

IMPROVING OPTICAL MARKER TRACKING PERFORMANCE

Excellent tracking performance can be obtained, even when using a relatively inexpensive "webcam," if you follow some basic guidelines:

- Calibrate your camera, as described in the previous section.
- Become familiar with your camera driver's control panel. Turn off any option that automatically computes exposure length, and instead set exposure length manually. Note that a long exposure will produce a blurred image when the camera and markers move relative to each other (especially if the motion is fast), resulting in poor tracking. Instead set your exposure length to be as short as practical.
- Since short exposures make for darker images, you will probably need to increase image brightness by shining additional light on the real world—either natural (pull up the shades) or artificial (turn on a bright lamp).
- Do *not* try to use your camera's "brightness" control to compensate for the darker image caused by a shorter exposure. You will instead create a lower contrast image that will not track as well.
- Focus your camera properly for the distances at which markers will be tracked. A smaller aperture will focus over a wider range, but admits less light, which is another reason to shine additional light on the real world.
- Explore your camera's (and computer's) tradeoffs between image resolution and number of frames captured/processed per second. For many recent USB 2 and IEEE 1394 (FireWire) cameras, you will get the best performance at 640×480 resolution.
- Make sure that your markers are as flat, clean, and wrinkle-free as possible. Print them on the heaviest stock your printer can handle, or attach them to cardboard or foamboard.
- Be aware of the specific requirements of the marker tracking software that you are using. For example, ALVAR will not recognize a marker whose border is not wholly visible (e.g., if even a small portion of the marker's border is obscured, it will not be tracked).

OPTICAL MARKER TRACKING WITH MULTIPLE CAPTURE DEVICES

Goblin XNA can use multiple capture devices for marker tracking. You simply need to add as many `IVideoCapture` implementations as you would like to the Scene class in order to capture from different physical cameras. Once you add them, you can

switch among the cameras to choose the one used for marker tracking by changing `Scene.TrackerVideoID`.

You should use identical video capture devices, since you can use only one calibration file for initializing ALVAR.

Note that there is also a `Scene.OverlayVideoID` property, which specifies which video capture device to use for overlaying the background, whilst `Scene.TrackerVideoID` specifies which video capture device to use for marker tracking. If these two properties are the same, which will typically be true, then the overlaid background image and the image used for tracking will be identical. However, there are some cases in which you may want them to be different. For example, if you want to use one camera only for tracking hand gestures with optical markers attached to a person's fingers or hand, and another camera for visualizing the world, then you can set `OverlayVideoID` to the camera used for visualizing the world, and `TrackerVideoID` to the camera used for tracking hand gestures. These two properties are set to the same ID when you add an `IVideoCapture` implementation to the Scene, and the video device added last is used by default.

These IDs are associated with the order in which you add your `IVideoCapture` devices to your scene graph using `Scene.AddVideoCaptureDevice(…)` method. This notion applies to the following section as well. Note that prior to v4.0, these IDs were associated with the video device IDs passed as the first parameter to `Scene.AddVideoCaptureDevice(…)` method.

STEREOSCOPIC AUGMENTED REALITY

When the virtual scene is rendered in stereo, the background image seen by each eye should be different. If you have more than two capture devices connected, then you can define which video (camera) image to render on the background for the left eye and which for the right eye by setting the `Scene.LeftEyeVideoID` and `Scene.RightEyeVideoID` properties. If you have only one capture device and want to simulate a stereo background, then you can set both `LeftEyeVideoID` and `RightEyeVideoID` to the same ID. Please see Tutorial 13 for an example of stereoscopic augmented reality using either Vuzix iWear VR920 or Wrap 920 head-worn displays.

For Wrap920 head-worn displays, you will need to calibrate the stereo separation and adjust the field of view for proper stereoscopic AR experience. First, you need to calibrate the camera intrinsic parameters for both left and right cameras using the *CameraCalibration* tool (see how to use it at codeplex site).

After the calibration, open the *StereoCameraCalibration.sln* file in *GoblinXNAv4.0\tools\StereoCameraCalibration*. Add the two (left and right) calibrated files to your project solution, and set their output setting to "Copy if newer". Open the *Setting.xml* file and change the *LeftCameraCalibration* and *RightCameraCalibration* values to be your calibrated file names. Please read the descriptions of other parameters in the setting file and change them if necessary. Print out the groundALVAR.pdf located in *GoblinXNAv4.0\tutorials\Tutorial8-Optical Marker Tracking* on 8.5"×14" legal paper (if you don't have 8.5"×14" paper, then print it on 8.5"×11" paper with your printing program's 'Actual size' option). Once everything is set, run the program.

Bring the printed marker array in the views of both cameras and press the 'Space' key to capture the image. If it succeeds, you will see a message on the screen indicating the number of valid images captured so far. If it fails, just try capturing from another angle and distance. Try to capture images from various angles and distances, similar to the way in which you would calibrate the camera intrinsic parameters.

Once you captured enough images to calculate the stereo separation parameters, the program will save the calibration file, and you will see five virtual cubes overlaid on top of the marker array. To make sure that the calibrated separation parameters are correct, check whether the virtual boxes shown on the left and right image appear at the same place on the physical marker array. If the calibration went well, you should see the virtual boxes in stereo without much difficulty.

Next, you will adjust the field of view of each camera to match the field of view of your corresponding eye. Make sure that you have set the *IsFullScreen* setting to true before performing this step. You will make separate adjustments for the left and right cameras. The goal is to compensate, at least partially, for any mismatch between the field of view of the camera lens and the field of view of the display. This task is made more difficult because the camera sensor may be angled relative to the display.

Starting from the left camera (make sure to cover your right eye so that you are looking through your left eye alone), use the 'Up' and 'Down' arrow keys to zoom in and out until the size of a physical object seen through the left eye camera and viewed on the left display matches the size of that object seen directly by your unaided eye. This is most easily done by positioning your head relative to an upright rectangular object such that the object is seen partially through the display, and partially by your unaided eye. Select an object that is at a distance typical of the distance at which the majority of object in your scene will appear. Note that the procedure described in this paragraph sets the size of the object as viewed through the display, but not its alignment.

Once you have adjusted the size, use the 'Left' and 'Right' arrow keys to set the horizontal alignment of the same physical object, to line up each of its left and right edges as seen through the display with the actual edges seen directly with your unaided eye. Similarly, you can use the 'PageUp' and 'PageDown' keys to set the vertical alignment of the object, to line up each of its top and bottom edges, as seen through the display with the actual edges seen directly with your unaided eye.

Note that adjusting zoom sets the size of the rectangular portion of each camera image that will be mapped to the display, and horizontal and vertical alignment effectively sets the horizontal and vertical position of that rectangle within the camera image. This means that if you adjust the rectangle so that it is not fully contained within the camera image, the pixels that fall outside the image will be undefined, and filled in with black pixels.

At this point, you may iteratively readjust zoom (using the 'Up' and 'Down' arrow keys), horizontal alignment (using the 'Left' and 'Right' arrow keys), and vertical alignment (using the 'PageUp' and ;PageDown' keys).

Once you're satisfied with the left eye adjustment, press the 'Enter' key to adjust the right eye in a similar fashion. When you're done with both eyes, press the 'Space' key to save the adjustment parameters.

In stereoscopic rendering, the 2D UI (anything rendered through the UI2DRenderer class, including all of the 2D GUI components, as described in the 2D GUI section) also needs to be rendered at different screen coordinates for each eye. You can shift the 2DUI coordinates by setting the `Scene.GlobalUIShift` (in pixels) property. The 2D UI will be shifted to the right on the left eye image by ⌊`Scene.GlobalUIShift`/ 2⌋ and shifted to the left on the right eye image by ⌊`Scene.GlobalUIShift`/ 2⌋. The amount that the 2D UI is shifted affects its apparent depth in the 3D scene. You may find that the most comfortable setting will place the UI at an apparent depth that matches the depth of the 3D objects (real and virtual) that you are viewing while using the 2D UI. Note that this might change from one part of your application to another.

PHYSICS ENGINE

A physics engine is required for realistic rigid body physical simulation. Each scene graph can have a physics engine, which you can assign by setting `Scene.PhysicsEngine` to the physics engine implementation you want to use. Once it is set, the physical simulation is initialized and updated automatically by the `Scene` class. A Geometry node can be added to the physics engine by setting `GeometryNode.AddToPhysicsEngine` to true. Each Geometry node contains physical properties for its 3D model, and the physics engine uses these properties to create an internal representation of the 3D model to perform physical simulation. If a Geometry node that is added to the physics engine is removed from the scene graph, then it is automatically removed from the physics engine.

If the transformation of a Geometry node that has been added to the physics engine is modified by one or more ancestor Transform nodes in the middle of physical simulation, then the transformation of the Geometry node is reset to the modified transformation composed of its ancestor Transform nodes. (Note: This assumes the `ModifyPhysicsObject(…)` function is implemented in the IPhysics implementation you are using. Currently, the HavokPhysics implementation does not implement this function.) However, this may cause unexpected behavior, since the physics engine will transport the 3D object, rather than applying a proper force to move the 3D object to the modified transformation.

A Marker node does not affect the Geometry node transformation in the physical simulation, since this is not the desired behavior in general. However, if you want to modify the transformation of a Geometry node in the physics engine other than by modifying the transformation of a Transform node or by the physical simulation, you can use the `NewtonPhysics.SetTransform(…)` function.

PHYSICS ENGINE INTERFACE

We define an interface, `IPhysics`, for any physics engine that will be used by our scene graph. This makes it possible for a programmer to implement his/her own physics engine and use it in Goblin XNA, provided that all of the methods required by the `IPhysics` interface are implemented. The `IPhysics` interface contains properties such as gravity and direction of gravity, and methods to initialize, restart, and update the physical simulation, and to add and remove objects that contain physical properties. Goblin XNA provides two default physics engine implementations, which wrap the Newton Game Dynamics 1.53 library and the Havok Physics library. Note that we do not support all of the functionality provided by these physics engines, but just enough to perform basic simulations.

PHYSICAL PROPERTIES

Before you can add an object that either implements or contains the `IPhysicsObject` interface (for example, the `GeometryNode.Physics` property contained in a Geometry node is an implementation of the IPhysicsObject), you must define the following physical properties:

- `Mass`. The mass of the 3D object.
- `Shape`. The shape that represents this 3D object (e.g., Box, Sphere, or Cylinder).
- `ShapeData`. The detailed information of the specified shape (e.g., each dimension of a Box shape).
- `MomentOfInertia`. The moment of inertia of the 3D object. (This can be computed automatically if not specified, but that will not guarantee the desired physical behavior.)
- `Pickable`. Whether the object can be picked through mouse picking.
- `Collidable`. Whether the object can collide with other 3D objects added to the physics engine.
- `Interactable`. Whether the object can be moved by external forces.
- `ApplyGravity`. Whether gravity should be applied to the object. There are other physical properties that do not need to be set. However, setting them can, for example, allow an object to have initial linear or angular velocity when it is added to the physics engine.

NEWTON MATERIAL PROPERTIES

In addition to the physical properties, a physics object can also have physical material properties such as elasticity, softness, and static and kinetic friction with another material (which can be either homogeneous or heterogeneous). Certain physics engines support material properties, and those that support material properties have default values for all physics objects added to the engine. If you want to modify the default material property values for interaction between certain materials, you can assign a material name to the physics object. For the NewtonPhysics implementation, the material name of a physics object is required and can be set in the `IPhysicsObject` interface (e.g., in a Geometry node, with the `GeometryNode.Physics.MaterialName` property). For those physics objects whose default material property values you do not want to modify, just leave the material name empty, which it is by default.

Once you have defined the material name, you can register a `NewtonMaterial` object to the physics engine by calling the `NewtonPhysics.AddPhysicsMaterial(…)` function. (Note: We do not expect

all physics engine to support all physics material properties, so we do not specify this method in the `IPhysics` interface. You can only add a physical material to a specific physics engine that supports it. The Newton physics engine supports all physical material properties, so you can add them.) Please see Tutorial 9 for a demonstration of how to define and add a `NewtonMaterial` object.

These are the material properties[3]:

- `Elasticity` (0.0f–1.0f): The larger the value, the less the vertical (relative to the colliding surface) energy is lost when two objects collide. This is the *coefficient of restitution*.
    - E.g., if elasticity is set to 1.0f, vertical energy loss is 0. If elasticity is set to 0.0f, all vertical energy is lost.
- `Static`, which represents *static friction* (0.0f–1.0f): The larger the value, the more horizontal (relative to the colliding surface) energy is lost when two objects collide.
    - E.g., if static friction is set to 0.0f, horizontal energy loss is 0. If static friction is set to 1.0f, all horizontal energy is lost.
- `Kinetic`, which represents *kinetic friction* (0.0f–1.0f): This should be less than the static friction for realistic physical simulation. The larger the value, the more horizontal energy is lost when two objects are sliding against each other.
- `Softness` (0.0f–1.0f): This property is used only when two objects interpenetrate. The larger the value, the more restoring force is applied to the interpenetrating objects. *Restoring force* is a force applied to make both interpenetrating objects push away from each other, so that they no longer interpenetrate.

NEWTON PHYSICS LIBRARY

The `NewtonPhysics` class implements the `IPhysics` interface using the Newton Game Dynamics 1.53 library, which is written in C++. We use the C# Newton Dynamics wrapper created by Flylio, with some bug fixes, in order to interface with the Newton library. In addition to the methods required by the `IPhysics` interface, we added several methods that are specific to the Newton library to perform the following functions:

- Apply linear velocity, angular velocity, force and torque directly to an `IPhysicsObject`.

[3] These descriptions are specific to `NewtonPhysics`, and may not apply if another `IPhysics` implementation is used.

- Specify physical material properties, such as elasticity, static and kinetic friction, and "softness" (`NewtonMaterial`).
- Pick objects with ray casting.
- Disable and enable simulation of certain physics objects.
- Provide collision detection callbacks.
- Set size of the simulation world.
- Provide direct access to Newton Game Dynamics library APIs (using `NewtonWorld` property and APIs wrapped in *NewtonWrapper.dll*).

Newton Game Dynamics has a notion of the size of the simulation world, which defines the volume in which the simulation takes place. The default size is 200×200×200, centered at the origin. If an object leaves the bounds of the simulation world, then the simulation of the object stops. You can modify the size of the simulation world by modifying the `NewtonPhysics.WorldSize` property.

`NewtonPhysics` automatically calculates some of the necessary properties of a physics object mentioned in the Physical Properties section if they are not set. If the `IPhysicsObject.ShapeData` property is not set, then it is automatically calculated using the minimum bounding box information associated with the 3D object. If the `IPhysicsObject.MomentOfInertia` property is not set, then it is automatically calculated using the utility function provided by the Newton Game Dynamics library.

Since the meanings of `IPhysicsObject.Collidable` and `IPhysicObject.Interactable` can be confusing, we list all four combinations of these two property values, and the meaning of the combinations[4]:

- Both `collidable` and `interactable` are set to true.
  The object collides with other collidable objects and reacts in response to physical simulation when force and/or torque are applied.
- Both `collidable` and `interactable` are set to false.
  The object still collides with other collidable objects, but does not react in response to physical simulation. This may seem strange, but once an object is added to the Newton Game Dynamics physics engine, it becomes collidable by default, and this cannot be changed except by using `IPhysicsMaterial` to specify specific materials that are not collidable with each other.
- `collidable` is set to true, but `interactable` is set to false.
  The object collides with other collidable objects, but does not react in response to physical simulation when force and/or torque are applied.

[4] These meanings are specific to NewtonPhysics, and may not apply if another IPhysics interface is used.

- `interactable` is set to true, but `collidable` is set to false.
  The object behaves as if both `collidable` and `interactable` are set to false. However, once force, torque, linear velocity, or angular velocity is applied, it starts behaving as if both `collidable` and `interactable` were set to true.

The addition and removal of any physics objects associated with Geometry nodes in our scene graph is handled automatically, so you should not call `AddPhysicsObject(…)` or `RemovePhysicsObject(…)` in your code for Geometry nodes added to the scene graph. However, if you decide to create your own class that implements the `IPhysicsObject` interface and want to add it to the physics engine for simulation, then you will need to take care of adding and removing that physics object in your code.

After a physics object is added to the physics engine, the transformation of the object is controlled based on the physical simulation. If you want to modify the transformation of the physics object externally, there are two ways to do so. One way is to call the `NewtonPhysics.SetTransform(…)` function if you want to set the transformation of the object yourself. If the physics object is associated with a Geometry node, then you can modify the transformation of the Transform node to which this Geometry node is attached (if any). In either case, this approach will transport the object instead of moving the object, so you may see unexpected, discontinuous behavior. An alternative is to call the `NewtonPhysics.AddForce(…)` and `NewtonPhysics.AddTorque(…)` functions. These functions will properly modify the transformation of the object; however, you may find it difficult to determine the exact force and torques that are needed to make the physics object have a specific transformation, if that is what you want. Nevertheless, we recommend using these two methods to modify the transformation of a physics object in the simulation externally, rather than setting the transformation explicitly.

Goblin XNA currently supports most of the capabilities of the original Newton Game Dynamics 1.53 library in `NewtonPhysics`. To perform more advanced simulation, you can directly access the full functionality of the original Newton Game Dynamics library using the `NewtonWorld` handler. Please see the API documentation for the `NewtonPhysics` class and the documentation of the original Newton library for details on how to directly access and use it. Most of the functions in the original Newton Game Dynamics library need a pointer to a `NewtonBody` instance. To get the `NewtonBody` pointer, you can use the `NewtonPhysics.GetBody(…)` method to access the pointer associated with the physics object used in Goblin XNA.

HAVOK PHYSICS LIBRARY

The `HavokPhysics` class implements the `IPhysics` interface using Havok physics (version hk710r1), which is written in C++. We wrapped important functionality in managed C (tools/HavokWrapper) and access the managed interfaces through `HavokDllBridge` in C#.

Havok requires information about the simulation world at the time of initialization, such as gravity, gravity direction, simulation type, and collision tolerance, so you need to pass a `HavokPhysics.WorldCinfo` structure when you instantiate the `HavokPhysics` implementation:

- `WorldSize`: Similar to Newton Game Dynamics, Havok also has a notion of the simulation world size, and any physics objects that go beyond the specified volume will not be simulated. The simulation volume is always a cube, so if you set WorldSize to be 150, then it means the volume is 150x150x150, centered at the origin. If you want to handle physics objects that go beyond this volume (e.g., delete them from the scene graph), then you can set the `HavokDllBridge.BodyLeaveWorldCallback` delegate function by calling the `HavokPhysics.SetBodyWorldLeaveCallback(…)` method.
- `HavokSimulationType`: Havok supports several simulation types, among which are `SIMULATION_TYPE_DISCRETE` and `SIMULATION_TYPE_CONTINUOUS`. For detailed explanations of these types, please read the original Havok documentation. If you do not want penetrations to occur, then you should use `SIMULATION_TYPE_CONTINUOUS`; otherwise, it is better to use `SIMULATION_TYPE_DISCRETE`, which is faster.
- `FireCollisionCallbacks`: By default, Havok does not fire any callbacks for collision events. If you want to handle collision events by setting `HavokObject.ContactCallback`, `HavokObject.CollisionStartCallback`, or `HavokObject.CollisionEndCallback` for your physics object, then you should set `FireCollisionCallbacks` to `true`.

For more detailed explanations, please see the original Havok documentation. We do not cover all of the initialization parameters supported by Havok, so if you need additional functionality, please modify the appropriate source code (HavokPhysics.cs, HavokDllBridge.cs, and HavokWrapper.cpp).

In addition to the regular physical properties specified in IPhysicsObject, there are several additional properties for each rigid body in Havok:

- `MotionType`: The motion type used to simulate the rigid body.
- `QualityType`: The quality of the collision result.
- `Friction`: The friction of the rigid body.
- `Restitution`: The restitution of the rigid body (identical to Elasticity in Newton).
- `ConvexRadius`: Havok creates a layer around each rigid body to prevent penetration between objects. The larger the convex radius, the less likely it is that objects will penetrate. However if the convex radius is too big, the gap between colliding objects will be visually obvious.
- `IsPhantom`: A phantom object does not collide with other objects, but can be used to detect whether another object entered or left the volume of the phantom object. (Make sure you set the `PhantomEnterCallback` or `PhantomLeaveCallback` function if you want to handle enter or leave events)

These additional properties are implemented in HavokObject, which extends PhysicsObject, so when you use HavokPhysics, you should assign HavokObject to GeometryNode.Physics (it will work without the assignment, but you will not be able to set Havok-specific parameters). The physical material properties in Havok are specified for each rigid body, rather than for each pair of rigid body types. Again, we do not cover all of the rigid body properties supported by Havok; therefore, for a detailed explanation of each property, please see the original Havok documentation.

One advantage of using Havok over Newton Game Dynamics is that one can move a static object (which, by definition, has infinite mass and does not move, such as the earth in many simplistic terrestrial simulations), so as to cause proper physical reactions between other objects in the simulation world, by simply giving the static object a new position and orientation. To do this, you need to set HavokObject.MotionType to `MotionType.MOTION_KEYFRAMED` for the static object you want to move around. Then use `HavokPhysics.ApplyKeyframe(…)` to move the static object to a desired location with the desired orientation. Note that if you do not call `HavokPhysics.StopKeyframe(…)`, the static object will keep moving even after it reaches the destination. However, if you keep setting the new location and orientation through `ApplyKeyframe(…)` for every frame, then you do not need to call `StopKeyframe(…)`. Call `StopKeyframe(…)` once you are done moving the object.

USER INTERFACE

Goblin XNA supports a set of common 2D graphical user interface (GUI) components that can be overlaid on the scene. All of the 2D UI rendered by Goblin XNA is drawn on the frontmost layer in the `SpriteBatch` class (with `layerDepth` set to `0`).

Note: Since there is no well-defined standard set of 3D GUI components, and 3D GUI component can be implemented using Geometry nodes combined with the physics engine, we decided not to provide a set at this time. However, we may include a set in a future release. We are also considering supporting texture-mapped 3D GUI components that use the texture of a live 2D GUI component.

2D GUI

In Goblin XNA, 2D GUI components are overlaid on the 3D scene. They can be set to be transparent, so that both the GUI and the 3D scene behind it are visible. The 2D GUI API is very similar to that of Java.Swing, including component properties and event handling (e.g., for button press actions). Please see Tutorial 3 for a demonstration of how to specify a collection of simple 2D GUI components. We currently support basic 2D GUI components that include panel, label, button, radio button, check box, slider, text field, and progress bar. (Future releases may include other components, such as combo box, text area, and spinner.)

Some of the 2D GUI components can display text. However, in order to display text, you will need to assign the font (`SpriteFont`) to use by setting the TextFont property for each component that displays text.

For 2D rendering, we support three layers through two callback functions: `Scene.RenderBeforeUICallback` and `Scene.RenderAfterUICallback`, and `UI2DRenderer` class. Our 2D rendering pipeline uses the following order:

1. RenderBeforeUICallback
2. Any 2D elements including texts, shapes, and textures that are rendered through the UI2DRenderer class (e.g., UI2DRenderer.WriteText(...), UI2DRenderer.FillRectangle(...))
3. RenderAfterUICallback

For example (as demonstrated in Tutorial 15), if you want to perform edge-detection on the video image and add the 2D lines on top of the video, but beneath the 2D UI elements, then you can implement this overlay functionality in a callback function and assign it to the `Scene.RenderBeforeUICallback` property.

BASE 2D GUI COMPONENT

All 2D GUI components inherit the properties and methods from the G2DComponent class, which inherits from the Component class. The individual 2D GUI class then extends this base class by either adding specific properties and methods or overriding the properties or methods of the base class. In order to find all of the properties and methods in the API documentation for a certain 2D GUI class, you will need to see its base class.

PANEL (CONTAINER)

Unlike Java.Swing, we do not have a "Frame" class, since XNA creates a window to hold the paintable area. Thus, the UI hierarchy in Goblin XNA starts from "Panel" instead of "Frame." Like Java.Swing, our "Panel" class, `G2DPanel`, is used to hold and organize other 2D GUI components such as `G2DButton`. However, we currently do not support an automatic layout system, so you will need to lay out the G2D components in the `G2DPanel` class yourself. Like Java.Swing, the bound (x, y, width, height) properties of each added G2D component are based on the `G2DPanel` to which they are added. Thus, if the `G2DPanel` has a bound of (50, 50, 100, 100) and its child `G2Dcomponent` has a bound of (20, 0, 80, 20), then the child G2D component is drawn from (70, 50). In addition to the bound property, the transparency, visibility, and enable properties affect child G2D components; for example, if the `visibility` of a `G2DPanel` is set to false, then all of its child G2D components will be invisible, as well.

LABEL

Like Java.Swing's JLabel, `G2DLabel` places an unmodifiable text label on the overlaid UI, and no event is associated with this class. The difference is that JLabel hides the text that exceeds the width of the bound; for example, if the bound is set to have width of 100, and the text length is 150, then the text part that overflows the bound (the remaining 50) will not be shown, but abbreviated with an ellipsis ("..."). In contrast, `G2DLabel` shows all of the text, even if some portion extends beyond the width of the bound.

BUTTON

Like Java.Swing's JButton, `G2DButton` is used to represent a clickable button. `G2DButton` has an action event associated with it, and it is activated when the button is pressed. You can add an action handler to `G2DButton.ActionPerformedEvent`, and the handler will get called whenever there is a button press action. In addition to directly pressing the button using a

mouse click, you can also programmatically press the button by calling the `G2DButton.DoClick()` function.

RADIO BUTTON

Like Java.Swing's JRadioButton, `G2DRadioButton` is used to represent a two-state (selected or unselected) radio button. The text associated with `G2DRadioButton` is displayed on the right of the radio button. Like the `G2DButton` class, you can add an action listener to the `G2DRadioButton` class, and the listener will get called whenever the radio button is pressed to switch to either the selected or unselected state. In addition to directly pressing the button using a mouse click, you can also programmatically press the button by calling the `G2DRadioButton.DoClick()` function.

Radio buttons are usually used in a group of radio buttons, with only one selected at time for single-choice selection. As in Java.Swing, you can use the `RadioGroup` class to do this. Simply add a group of radio buttons to a `RadioGroup` object, and set one of the radio buttons to be selected initially.

CHECK BOX

Like Java.Swing's JCheckBox, `G2DCheckBox` is used to represent a two-state (checked or unchecked) check box. Check boxes are very similar to radio buttons. The only difference is that check boxes are usually used for multiple-choice selection.

SLIDER

Like Java.Swing's JSlider, `G2DSlider` is used to select a value by sliding a knob within a bounded interval. The slider can show both major tick marks and minor tick marks between them, as well as labels, by setting `G2DSlider.PaintTicks` or `G2DSlider.PaintLabels`, respectively, to `true`. You will also need to set `G2DSlider.MajorTickSpacing` and `G2DSlider.MinorTickSpacing`, respectively, to see major tick marks and minor tick marks. You can modify the slider value either by sliding the knob using the mouse or programmatically setting `G2DSlider.Value`. For a label, we recommend that you use a smaller font than the other UI fonts.

The event handler for `G2DSlider` is `StateChanged`, and you can add an implementation of this delegate function to `G2DSlider.StateChangedEvent`. Then, the implemented delegate function will get called whenever the value of the slider changes.

TEXT FIELD

Like Java.Swing's JTextField, `G2DTextField` displays a text string from the user's keyboard input. The text field needs to be focused in order to receive the key input. In order to focus on the text field, the user needs to click within the bounds of the text field using the mouse.

The event handler for `G2DTextField` is `CaretUpdate`, and you can add an implementation of this delegate function to `G2DTextField.CaretUpdateEvent`. Then, the implemented delegate function will get called whenever the caret position changes.

PROGRESS BAR

Like Java.Swing's JProgressBar, `G2DProgressBar` displays an integer value within a bounded interval. A progress bar is typically used to express the progress of some task by displaying the percentage completed, and optionally, a textual display of the percentage. In order to show the textual display of the percentage, you need to set `G2DProgressBar.PaintString` to true. You can also change the color of the progress bar or the textual display by setting `G2DProgressBar.BarColor` or `G2DProgressBar.StringColor`, respectively.

In addition to the normal mode that displays the percentage completed, `G2DProgressBar` also has a mode called "indeterminate." Indeterminate mode is used to indicate that a task of unknown length is running. While the bar is in indeterminate mode, it animates constantly to show that some tasks are being executed. You can use indeterminate mode by setting `G2DProgressBar.Indeterminate` to `true`.

LIST

Like Java.Swing's JList, `G2DList` displays a list of items within a bounded region, and each of the listed items is selectable. The selection method can be single, multiple with single interval, or multiple with multiple intervals. The list data model, selection model, and cell renderer can be easily replaced with custom implementations. Note that `G2DList` renders the entire part of each cell, even if some portion extends beyond the width or height of the bound.

SPINNER

Like Java.Swing's JSpinner, `G2DSpinner` allows the user to select an item defined by a `SpinnerModel` from an ordered sequence. `G2DSpinner` provides a pair of tiny arrow buttons for stepping through the elements of the sequence. The user can also

use keyboard up/down arrow keys to cycle through the elements. The value in the spinner is not directly editable.

MORE COMPLEX COMPONENTS

There are few more complex components (e.g., `G2DMediaControl`, `G2DSuggestField`, and `G2DWaitBar`) that combine two or more basic UI components or extend the original functionalities of other UI components to perform more advanced user controls. These components demonstrate how you can extend the existing components to create customized advanced user controls, and can be found in the `GoblinXNA.UI.UI2D.Fancy` package.

EVENT HANDLING

Unlike Java.Swing, Goblin XNA handles events using delegate functions and *event* properties associated with each G2D class. There are different types of delegate functions, and you need to implement the right delegate function for a specific event. Then, you can register the implemented delegate function to a G2D component through one of its *event* properties, and when the event occurs, the registered delegate function will get called.

For example, you can implement a function that has the same parameter set as the `ActionPerformed` delegate, and add the implemented function to G2DButton's `ActionPerformedEvent` *event* property. Then, whenever the button is pressed, the implemented delegate function will get called.

SHAPE DRAWING

Goblin XNA provides several functions for drawing simple 2D shapes such as line, rectangle, circle, and convex polygon. Similar to Java2D, there are draw functions and fill functions. These functions can be found in the `GoblinXNA.UI.UI2D.UI2DRenderer` class.

GUI RENDERING

The user interface is rendered by the `GoblinXNA.UI.UIRenderer` class. In order to render G2D components, you need to add them to this class by calling `Add2DComponent(…)`. Even though you can call the `RenderWidget(…)` method for each individual G2D components, it is not recommended to do so. You should simply add them to `UIRenderer` and let this class take care of the rendering. Also, you should only add the topmost G2D components to `UIRenderer`.

3D TEXT RENDERING

Goblin XNA supports 3D text rendering in outlined, filled, or extruded style using a slightly modified version of the Nuclex library. 3D text rendering functions can be found in the `GoblinXNA.UI.UI3D.UI3DRenderer` class, and an example is provided in **Tutorial 9**. In addition, you can use the `Text3D` class in the `GoblinXNA.Graphics` package that extends `PrimitiveModel`, and you can associate the 3D text with a `GeometryNode` for easier transformation control and physics simulation.

SOUND

We created a wrapper for the XNA audio engine to provide an easier interface for playing 2D and 3D audio. Before you can play audio, you first need to initialize the audio engine by calling the `Sound.Initialize(String xapAssetName)` function. (Sound is a static class, so you do not need to call the constructor.) The XACT project file (*.xap*) can be created using the "Microsoft Cross-Platform Audio Creation Tool" that comes with XNA Game Studio 4.0. To learn how to compile an XACT project file, see http://msdn.microsoft.com/en-us/library/ee416205(VS.85).aspx. For a 3D sound effect, you will need to create proper XACT Runtime Parameter Controls (RPC), such as volume attenuation based on distance or the Doppler effect, and associate them to the 3D sound effect using the audio creation tool. If these RPCs are not attached to the sound, the sound will be 2D. Please see http://msdn.microsoft.com/en-us/library/ee416009(VS.85).aspx for a description of how to create and associate sounds with RPCs. Once the XACT project file is created, add it to your "Content" directory through the solution explorer, and pass the asset name to this initialization function.

Now, you can play any 2D or 3D audio object by calling `Sound.Play(String cueName)` or `Sound.Play3D(String cueName, IAudioEmitter emitter)`, respectively. The cue name is the one you defined when you created your XACT project file, but not the actual audio (.wav) filename you want to play. You can control the audio (e.g., stop, pause, and resume) and get the audio status (e.g., whether it is created or playing) using the "Cue" object returned from both of the "Play" functions. However, if you force the audio to stop or the audio finishes playing, Goblin XNA automatically disposes of that audio object, so you will not be able to play the same sound again using the returned "Cue" object. In order to play it again after either you force it to stop or it finishes playing, you should call the "Play" function again to get a new "Cue" object to control it.

SHADERS

Instead of using a set of fixed-pipeline functions to render the 3D object and the scene, XNA Game Studio uses a programmable 3D graphics pipeline with the ability to create customized vertex shaders and pixel shaders. Even though this gives the programmer more power and flexibility in determining how the 3D objects in the scene are rendered, it means that you will need to create your own shader to draw even a single triangle. Since many programmers new to the idea of a shader language do not want to spend time learning how to write shaders, we provide a set of shaders in Goblin XNA, including a simple effect shader that can render many kinds of simple objects, a more advanced general shader, a particle shader, and a shadow mapping shader.

If you wish to create your own shader or use another shader, you will need to implement the `IShader` interface. (Tutorial 11 shows you how to create your own shader and incorporate it into Goblin XNA.) Goblin XNA stores global and local Light nodes in the order in which they are encountered in the preorder tree traversal of the scene graph. The implementer of the shader interface determines which of the light sources in these Light nodes are passed to the shader. This will typically be important if the shader has a limit on the number of light sources that it supports, and the Goblin XNA user has created a scene graph that includes more than this number of global light sources and local light sources that can affect an object being illuminated. The shader interface code determines which lights take priority. For example, a shader interface may give priority to global lights over local lights, or local lights over global lights, if it will not pass all of the lights to the shader.

SIMPLE EFFECT SHADER

The simple effect shader (`GoblinXNA.Shaders.SimpleEffectShader`) is used by default for rendering the `GoblinXNA.Graphics.Model` object in the Geometry node and bounding boxes. However, you can always replace it with a different shader if you prefer by setting the `Model.Shader` property. The simple effect shader uses the `BasicEffect` class provided by XNA Game Studio with the material properties and lighting/illumination properties associated with the 3D models.

One limitation of the `BasicEffect` class is that it can only use three light sources, all of which must be directional lights. Thus, if the `SimpleEffectShader` is used for rendering an object (which it is by default), then any lights that are not directional lights are ignored. If there are more than three light sources, then local light sources take precedence over global light sources.

DIRECTX SHADER

The DirectX shader (`GoblinXNA.Shaders.DirectXShader`) implements the fixed-pipeline lighting of DirectX 9. It is capable of rendering point, directional, and spot light sources, with no limitation on the number of light sources. The exact equations used for each light type can be found at http://msdn.microsoft.com/en-us/library/bb174697(VS.85).aspx. To use the DirectX shader, you need to add the *DirectXShader.fx* file located in the */data/Shaders* directory to your project content.

OTHER SHADERS

In addition to `SimpleEffectShader` and `DirectXShader`, we also included `ParticleShader`, which is a modified version of the shader used in the Particle 3D tutorial on the XNA Creator's Club website; `NormalMapShader`, which supports bump maps and environment maps; and `ShadowMapShader`, which is a modified version of the shadow-mapping shader provided in the XNA Racing Game Starter Kit.

`ParticleShader` is used to render particle effects in the Particle nodes. If you want to use your own particle shader, you can set `ParticleEffect.Shader` for any particle effect class that inherits `ParticleEffect` class. In order to use particle effects, you will need to add the *ParticleEffect.fx* file located in the */data/Shaders* directory to your project content.

`NormalMapShader` is used to render 3D models with bump map textures and environmental reflections. To assign bump map textures or environmental map texture for reflections, you need to use `NormalMapMaterial` instead of the regular `Material` class. The shader file, *NormalMapping.fx*, is located in the */data/Shaders* directory.

`MultiLightShadowMap` and `SimpleShadowShader` are used to render shadows cast on 3D objects in the scene. To display shadows, you need to set `Scene.EnableShadowMapping` to `true`, as well as set `Model.ShadowAttribute` for each 3D model you want to cast or receive shadows. (See Tutorial 8 for an example.) In order to use shadow mapping, you will need to add the *MultiLightShadowMap.fx* and *SimpleShadowMap.fx* files located in the */data/Shaders* directory to your project content. To make the shadow render properly, you may need to tweak the `DepthBias` properties of `MultiLightShadowMap` for any of the light types (e.g., directional, point, or spot) you use.

NETWORKING

XNA Game Studio supports networking functionality specific to games, but it requires that the user log in to the XNA Live server, which can only connect to other machines through the XNA "lobby" system. While this works well for many kinds of games, it can be cumbersome if the user simply wants to connect a set of machines with known IP addresses or host names, and communicate efficiently among them, and will be unreliable if reliable internet access is not available. Therefore, Goblin XNA includes its own network interfaces that can be used for any application.

To use the Goblin XNA networking facility, you first need to enable networking by setting `State.EnableNetworking` to `true`. Next, you need to specify whether the application is a server or a client by setting `State.IsServer` to `true` or `false`. Then, you need to define what implementation of network server (defined by the `IServer` interface) or client (defined by the `IClient` interface) you want to use. You can either create your own implementation of `IServer` and `IClient` using the network API of your choice, or you can use the Goblin XNA implementations of `IServer` and `IClient`, which use the Lidgren network library (the currently supported version is gen3 2011-2-19): `LidgrenServer` and `LidgrenClient`. Finally, you need to define what implementation of network handler you want to use defined by the `INetworkHandler` interface. We implemented `NetworkHandler` class as a default handler.

You will need to assign the `IServer` or `IClient` implementation to the `INetworkHandler` implementation, and then assign this handler to `Scene.NetworkHandler`. When you assign `IServer` or `IClient` to `NetworkHandler`, the `NetworkHandler` class will automatically call the `IServer.Initialize()` function or the `IClient.Connect()` function. If you want to set certain property values (e.g., `IClient.WaitForServer` for a client) to take effect before initializing the server or connecting to a server from the client, you should assign the `Scene.NetworkServer` or `Scene.NetworkClient` properties after setting the `IServer` or `IClient` properties. For `LidgrenServer` and `LidgrenClient` implementations, you can set Lidgren-specific configurations through the `NetConfig` property. Now, you are ready to send or receive any implementations of `INetworkObject`. Please see Tutorial 10 for a demonstration of a simple client-server example that transmits mouse press information.

SERVER

A server implementation should implement the Goblin XNA `IServer` interface. Even though you can call the broadcast or receive functions manually to communicate between clients, Goblin XNA automates this process through the `INetworkObject`

interface. (See the Network Object section for details.) We recommend that you send or receive messages by implementing the `INetworkObject` interface and add it to the scene graph using the `Scene.NetworkHandler.AddNetworkObject(…)` function.

If you want to perform certain actions when there is a client connection or disconnection, you can add `HandleClientConnection` or `HandleClientDisconnection` delegates to the `IServer.ClientConnected` or `IServer.ClientDisconnected` event.

CLIENT

Any client implementation should implement the Goblin XNA `IClient` interface. Again, you can manually communicate with the server, but we recommend using the `INetworkObject` interface. (See the Network Object section for details.) By default, if the server is not running at the time the client tries to connect to the server, the connection will fail and the client will not try to connect again. In order to force the client to continue trying to connect to the server until it succeeds, you will need to set `IClient.WaitForServer` to `true`. You can also set the timeout for the connection trial (`IClient.ConnectionTrialTimeOut`), so that it will not keep trying forever.

NETWORK OBJECT

The `INetworkObject` interface defines when or how often certain messages should be sent over the network, and how the messages should be encoded by the sender and decoded by the receiver. For any objects that you want to transfer over the network, you should implement `INetworkObject`, and add your object to the scene graph using the `Scene.NetworkHandler.AddNetworkObject(…)` function. Make sure you make the `INetworkObject.Identifier` unique relative to other network objects you add.

Once it is added to the scene graph, packet transfer is controlled by either `INetworkObject.ReadyToSend` or `INetworkObject.SendFrequencyInHertz`. The data sent is whatever is returned from your `INetworkObject.GetMessage()` function, and the received data is passed to your `INetworkObject.InterpretMessage(…)` function. If you want to send the packet at a specific time, then you should set `INetworkObject.ReadyToSend` to `true` at the specific time, and once the packet is sent, `INetworkObject.ReadyToSend` will be automatically set back to `false` by the scene graph. If you want to send the packet periodically, then you should set `INetworkObject.SendFrequencyInHertz`, which defines how

frequently you want to send in Hz (e.g., setting it to 30 Hz means to send 30 times per second). Message transmitting and receiving is processed during each `Scene.Update(…)` call. In case you do not want to have the packet sent for some period, even if either `INetworkObject.ReadyToSend` is set to `true` or `INetworkObject.SendFrequencyInHertz` is set to other than 0, you can set `INetworkObject.Hold` to `true`. As soon as you set it back to `false`, the scene graph will start processing the packet transfer.

You can also control whether the packets should always be transferred in order or can be sent out of order, by setting `INetoworkObject.Ordered`, and whether the receiver is guaranteed to receive the transferred packets by setting `INetworkObject.Reliable`.

Note that messages are transferred as arrays of bytes. Our `GoblinXNA.Helpers.ByteHelper` class provides several utility functions to perform tasks such as concatenating bytes and converting bytes to or from other primitive types. (The `BitConverter` class also provides many conversion functions.)

DEBUGGING

We currently support text-based debugging (with text either displayed on the screen or written to a text file), bounding-box–based graphical debugging, and a graphical scene graph display (which can be found under the *GoblinXNA/tools* directory.)

SCREEN PRINTING

Since Goblin XNA is a graphics package, you may want to print debugging information on the screen, instead of on the console window. We support printing to the screen in two ways. One general way is to use `GoblinXNA.UI.GUI2D.UI2DRenderer` to draw a text string using a specific font on the screen. Another way is to use the `GoblinXNA.UI.Notifier` class, which is designed specifically for debugging purposes. (Please see Tutorial 9 for a demonstration.) Here, we provide information on how to use `GoblinXNA.UI.Notifier`:

- To display messages (especially for debugging) on the screen instead of on the console window, pass text strings to `Notifier.AddMessage(…)`.
- To display messages added to the `Notifier` class, set `State.ShowNotification` to `true`.
- Display location can be changed by modifying the `Notifier.NotifierPlacement` enum. The choices are upper-left corner, upper-middle, upper-right corner, lower-left corner, lower-middle, lower-right corner, or custom location. The default location is upper-right. If custom location is chosen, then you also need to set the `CustomStartLocation` and `CustomAppearDirection` properties.
- Displayed text messages start fading out after `Notifier.FadeOutTime` milliseconds, and eventually disappear. The default `FadeOutTime` is set to `1` (never fade out).

In addition to your custom debugging message, we support printing out FPS (frames-per-second), and the triangle count of currently visible objects in the frustum. (Note that the triangle count actually includes objects near the boundary of the frustum even if they are not visible.) To enable display of this information, set `State.ShowFPS` and/or `State.ShowTriangleCount` to `true`.

LOGGING TO A FILE

If you prefer to print out debugging messages in a text file, you can use the `GoblinXNA.Helpers.Log` class. The Log class can help you write a log file that contains log, warning, or error information, as well as logged time for simple runtime error checking. By default, the log file will be created in the same directory as the

executable file. (You can change the file location by using the configuration file, as explained in the section on Configuration Files.) When you pass a text message to the `Log` class, you can also define the severity of the message in the second parameter of the `Log.Write(…)` method. If you do not set the second parameter, the default is `LogLevel.Log`. The `Log` class has a notion of print level, which defines the level of severity that should be printed. By default, the print level is set to `LogLevel.Log`, which means all of the logged messages will be printed to the log file. There are three levels, `Log`, `Warning`, and `Error`, in increasing order of severity. The print level is used to define the lowest severity level to print; severity levels at or above the print level will be printed; for example, if print level is `Warning`, then messages with severity level `Warning` and `Error` will be printed. You can modify the print level by changing `State.LogPrintLevel`. Logged messages will also be displayed on the screen if both `Log.WriteToNotifier` and `State.ShowNotification` are set to `true`.

MODEL BOUNDING BOX AND PHYSICS AXIS-ALIGNED BOUNDING BOX

For a 3D model, we support the capability of displaying its bounding box, as well as its axis-aligned bounding box acquired from the physics engine (which corresponds to the physics model used to represent the 3D model). You can use the bounding boxes to debug unexpected model behavior.

To display the bounding box of a model, set `Model.ShowBoundingBox` to `true`. You can also change the color of the bounding box and the shader used to draw the bounding box by setting `State.BoundingBoxColor` and `State.BoundingBoxShader`, respectively.

To display the axis-aligned bounding box, set `Scene.RenderAxisAlignedBoundingBox` to `true`. (If you are not using the physics engine for physical simulation, then there is no reason to display the axis-aligned bounding box.)

MISCELLANEOUS

SETTING VARIABLES AND GOBLIN XNA CONFIGURATION FILE

Setting variables can be loaded at the time of Goblin XNA initialization. The third parameter of `State.InitGoblin(…)` specifies an XML file that contains setting variables. For example, if a model *(.fbx)* is not added directly under the Content folder, Goblin XNA does not know where to find it. Thus, you need to specify the directory that contains the models in the setting file. The same is true for fonts, textures, audio, and shaders. Goblin XNA will generate a template setting file (*template_setting.xml*) that contains all of the setting variables used by Goblin XNA if you leave the third parameter as an empty string. Please see the generated template file for detailed descriptions of each setting variable.

You can also add your own setting variable to this XML file, such as:

```
<var name="SceneFile" value="scene.xml">
```

You can then retrieve the values associated with these setting variables by using the `State.GetSettingVariable(String name)` function. This is useful if you do not want to hard-code certain values in your program and be able to modify them from an XML file. As noted in the template setting file, you can also choose to remove any of the existing setting variables you do not need; for example, if all of the resource files are directly stored under the "Content" directory, then you do not need any of the "…Directory" setting variables.

PERFORMANCE

Goblin XNA takes advantage of multi-core CPU machines by multi-threading certain operations to speed up rendering. However, if your machine has a single-core CPU, using multi-threading may result in noticeably worse performance than not using multi-threading. Therefore, you may want to set `State.ThreadOption` to thread only specific operations (e.g., marker tracking or physics simulation). Note that if you try to multi-thread more operations than the number of cores your machine supports, Goblin XNA may run *slower,* rather than faster. Also, if you multi-thread marker tracking, the tracking of the 3D models may lag behind the video image due to the asynchronous relationship between tracking and rendering.

INDEX

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y