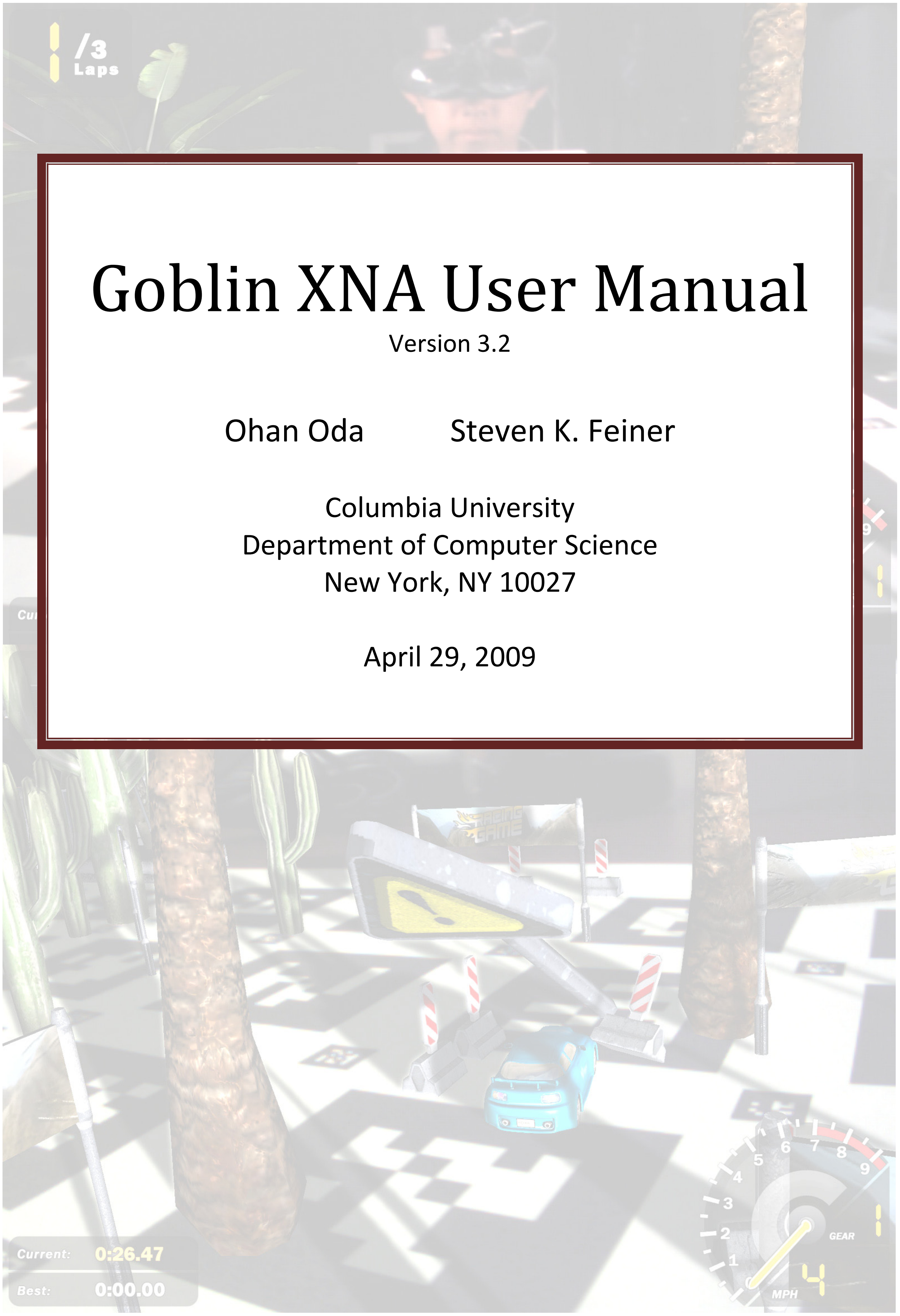

Goblin XNA User Manual

Version 3.2

Ohan Oda Steven K. Feiner

Columbia University
Department of Computer Science
New York, NY 10027

April 29, 2009

Table of Contents

0. Introduction

Goblin XNA is an open-source platform for research on 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 3.0. Goblin XNA uses a scene graph to support 3D scene manipulation and rendering, mixing real and virtual imagery. 6DOF (six-degrees-of-freedom) position and orientation tracking is accomplished using the ARTag marker-based camera tracking package with DirectShow or PGRFly (for Point Grey cameras), InterSense hybrid trackers, and the Vuzix iWear VR920 orientation tracker. Physics is supported through the Newton Game Dynamics 1.53 library, and networking through the Lidgren library. Goblin XNA also includes a 2D GUI system to allow the creation of classical 2D interaction components.

Like any program written using XNA Game Studio, a program that you write for Goblin XNA will be developed using Microsoft Visual Studio.

Goblin XNA is funded in part by a generous gift from Microsoft Research.

1. Getting Started

Begin by downloading all necessary dependencies and Goblin XNA itself, as described in the separate Installation Guide (InstallationGuide.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.

1.1 Using a project template

1. Open up the Tutorials.sln file in the GoblinXNA/tutorials directory by double-clicking it.
2. You will see thirteen tutorials. If 'Tutorial1—Getting Started' is not selected as a start up 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.*
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.

** 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 actually start.*

*One way to make your program start up faster (in Visual Studio 2008 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 actually runs.

1.2 From scratch

1. Open Visual Studio 2008, 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 3.0'. Then select 'Windows Game (3.0)'. Click '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 *GoblinXNA/bin* directory, as instructed in InstallationGuide.pdf. Since all of the other references required for Goblin XNA are also stored in the same directory as GoblinXNA.dll, Visual Studio will automatically copy those into your bin folder along with GoblinXNA.dll, so you do not need to add them to your 'References' section. 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 to your bin directory or add Newton.dll to your project and set the property option 'Copy to Output Directory' to 'Copy if newer'. If you want to use the ARTag marker tracker library, you will need to do the same steps for ARTagWrapper.dll and the marker configuration file (.cf file). These dlls *cannot* be added to the 'References' section because they are not managed dlls.

2. 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

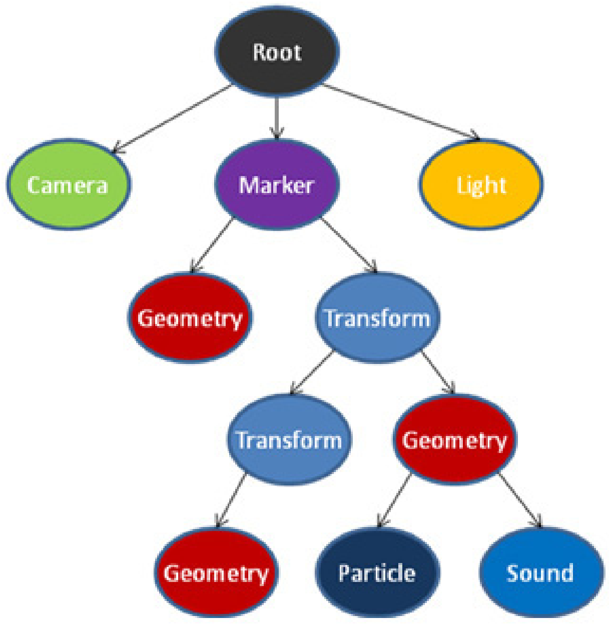

Figure 1. Goblin XNA scene graph hierarchy.

An example scene graph hierarchy is illustrated in Figure 1. 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) that 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.

2.1 Node types

2.1.1 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. 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 Section 5 for details on the supported physics engine.)
- Occlusion properties, which determine 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 through augmented reality) with which the virtual object interact.
- Network properties, which determine how geometry information is transferred over the network. (See Section 8 for details on supported networking.)

Geometry nodes are transformed by physical simulation, as described in Section 5. 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 2 accomplish the same thing: Both the Sound and Geometry nodes will be influenced by the Transform node.

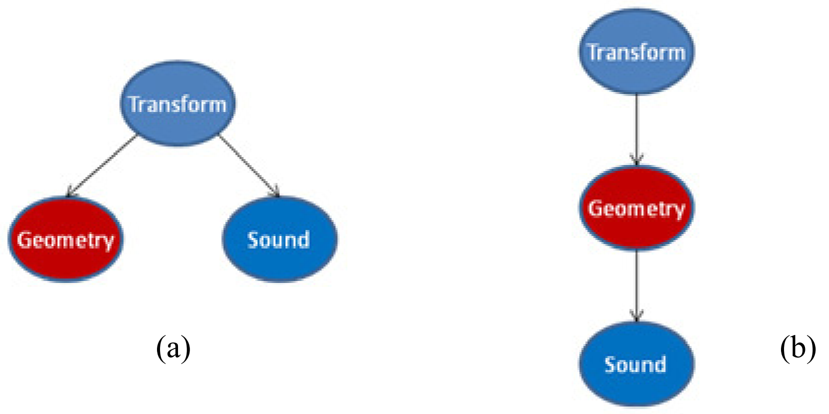

**Figure 2. (a) Sound is not affected by Geometry node transformation.
(b) Sound is affected by Geometry node transformation.**

However, if physical simulation is being performed (see Section 5), then the Sound node will be transformed *differently* in the two hierarchies, since the Geometry node's transformation is affected not only by the Transform node, but also by the physical simulation. For example, in Figure 2(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 2(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.

2.1.2 Transform

A Transform node modifies the transformation (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 assign each transformation component (translation, rotation, and scale) separately, causing the composed transformation matrix to be computed automatically from these values. An alternative is to directly assign the transformation matrix. Note that the *last* approach used determines the transformation. For example, if you assign the translation, rotation, and scale 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 translation, rotation, and scale, you need to assign a new value to one of these components, causing the node's transformation to once again be composed from the translation, rotation, and scale.

2.1.3 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 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 simplified theatrical light can be modeled as a spot light.

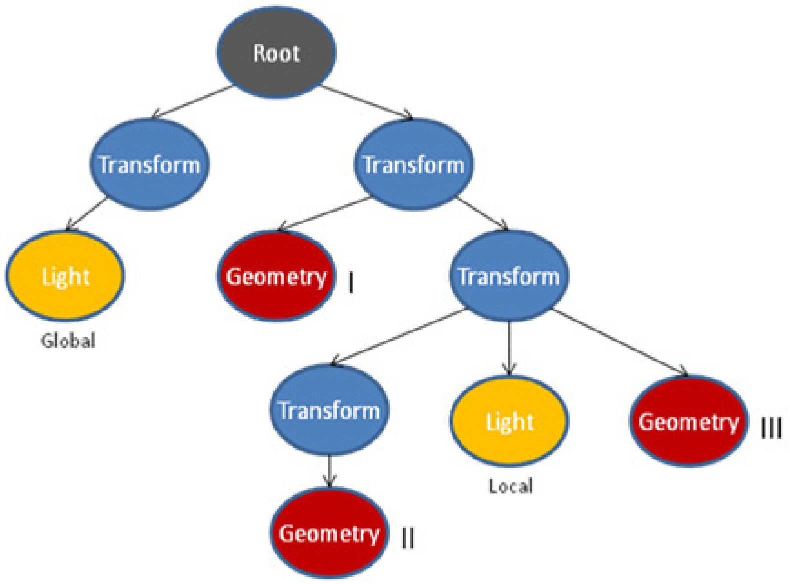

Figure 3. Global and local Light nodes.

A Light node can contain multiple light sources and an ambient light color, and the entire node can either be global or local. (All of the light sources contained in a Light node will have the same global or local setting.) 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 3, 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. Furthermore, note that if the global Light node in Figure 3 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.

2.1.4 Camera

A Camera node defines the position and visible *frustum* of the viewer (the 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 4. The initial view direction is toward the $-Z$ direction with an up vector of $+Y$. The Camera node's rotation property modifies this view direction by appling the given rotation to the initial view direction. You can create a view frustum that is a regular pyramid by assigning values to these parameters, causing the view and projection matrices to be computed automatically. Alternatively (typically 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.

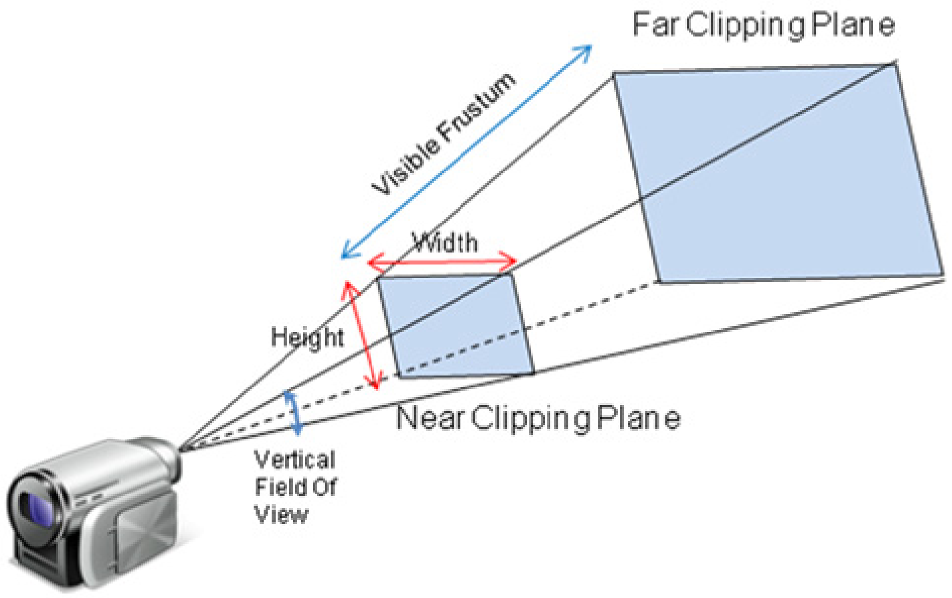

Figure 4. Camera geometry.

2.1.5 Particle

A Particle node contains one or more particle effects, such as fire, smoke, explosions, 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 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*.)

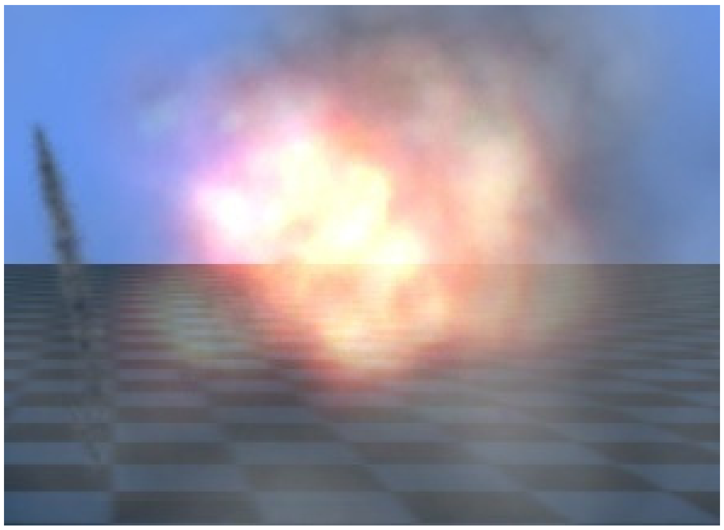
Figure 5. Explosion particle effect. (Courtesy of XNA Creators Club.)

2.1.6 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 of an array of one or more optical fiducial markers. Before you can use a Marker node, you will need to initialize the marker tracker and video capture devices, as explained in Sections 4.1–4.2. Then, you can create a Marker node to track a specific marker array defined in the configuration file. 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 supported by Goblin XNA or create your own smoothing filter by implementing the ISmoother interface. The following explanation applies only if you use the DESSmoother class in Goblin XNA.

A 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 poses tracked recently. (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 frequently, 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 less frequently, we recommend that you use a lower smoothing alpha.

2.1.7 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 Section 7 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 have a 3D sound effect, 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 Section 7 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).`

2.1.8 Switch

A Switch node is used to select a single child node to traverse among its child nodes. The `SwitchID` property denotes the index in the 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.

NOTE: *The Switch node has not been tested rigorously in the current release of Goblin XNA, and may not work as expected for certain cases.*

2.1.9 LOD

An LOD (Level of Detail) node is used to select only one model to be rendered from a list of models with different level of details. This node extends the Geometry node, but 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.

2.1.10 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. (You can obtain 6DOF device names from the GoblinXNA.Device.InputMapper class.) 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.

2.2 Scene

The Scene class is our 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 won't 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 point, 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. (A tutorial will be added later to show how you can create multiple viewports.)

Goblin XNA traverses and renders the scene graph automatically. Therefore, even though the scene graph has `Update(…)` and `Draw(…)` functions, you should *not* call them yourself. (They are called in the `base.Update(…)` and `base.Draw(…)` functions in your main Game application.)

2.3 Stereoscopic graphics

We support stereoscopy through the StereoCamera class when an appropriate stereo device is present. Currently, the only stereo device support is the Vuzix iWear VR920 head-worn display. 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: The only additional property you will need to set is `InterpupillaryDistance`, which defines the distance between the user's left and right eyes (cameras). Once the camera is set to stereo, the Scene class renders the scene for left and right eye in turn. To correctly display the left and right eye scenes to the corresponding eyes using the Vuzix iWear VR920, you will need to synchronize the device after rendering each scene. Please see **Tutorial 13** to see how you can correctly set up stereoscopic rendering for the iWear VR920. Note that a stereo application should be set to run in full-screen mode. Otherwise, the synchronization will not work properly and you will see the left and right eye views swapping frequently.

3 6DOF and 3DOF Tracking and Input Device Abstraction

We currently support 6DOF (six–degrees-of-freedom) position and orientation tracking using the ARTag optical marker system and InterSense hybrid trackers, and 3DOF (three-degrees of freedom) orientation tracking with the Vuzix iWear VR920 orientation tracker. The ARTag interface is provided though the Marker node, as described in Section 2.1.6, as well as the Scene class. The InterSense trackers and Vuzix orientation tracker (iWearTracker), however, is interfaced through the InputMapper class, which provide hardware device abstractions for various input devices. (Since ARTag is not really a hardware device, it is not interfaced through InputMapper.) Other input devices currently supported through this class are: mouse, keyboard, a combination of mouse and keyboard that simulates a 6DOF input device, and GPS. All of these device classes are singleton classes, so you need to use their Instance properties to access them.

3.1 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 API documentation for the InputMapper class.

3.1.1 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.. Please refer to Section 11.1 for details on how to write or modify a configuration file. If this network configuration is not found, only a direct connection will be attempted.

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.

3.1.2 Vuzix iWear VR920 orientation tracker

Once you install the appropriate Vuzix driver and software for the iWear VR920, 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.

3.1.3 Combination of mouse and keyboard

A combination of the mouse and keyboard device is used to support a simulated 6DOF input device in which the mouse controls the orientation and the keyboard controls the translation. To modify the orientation, hold the right mouse button and drag the mouse around on the screen. To modify the translation, use the W (forward), S (backward), A (left), D (right), Z (up), and X (down) keys. Users can change the keys that modify the translation and movement, yaw rotation, and pitch rotation speed by modifying the properties in the GenericInput class. This class is intended for simple 3D navigation during debugging. We do not recommend using this class for navigation in an end-user application.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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).

4 Augmented Reality

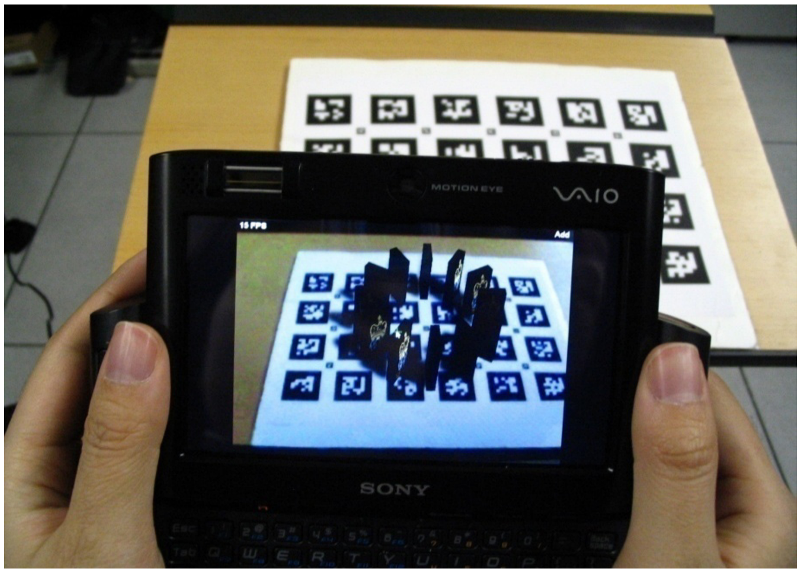

Figure 6: Virtual dominoes are overlaid on a real video image using Goblin XNA.

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 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 6. Two important issues that Goblin XNA addresses for video see-through AR are combining rendered graphics with the real world image and pose (position and orientation) tracking for registration. To support video devices for capturing the real image, we support two different types of video decoding libraries: DirectShow and PGRFly (from Point Grey Research, and used only with Point Grey cameras).

4.1 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 two different implementations of the IVideoCapture interface: DirectShowCapture and PointGreyCapture. If you are using a standard web camera, then DirectShowCapture is suitable. If you are using a Point Grey camera (e.g., a Firefly or Dragonfly), then PointGreyCapture is suitable. If you want to use another video decoding library, such as OpenCV, 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(…)`. Also, 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, you can get the video image in Bitmap format for DirectShowCapture by calling the `GetBitmapImage()` function.) Alternatively, you can get the video image by either accessing the `ImagePtr` property or calling the `GetImageTexture(…)` function, but you will need to be careful. This property and function are used internally by the Goblin XNA framework, so it may cause unexpected behavior. If the optical marker tracker is set for pose tracking, then `ImagePtr` is always valid and safe to access, but if it is not set for pose tracking, then `ImagePtr` is always an invalid pointer. The format of ImagePtr is defined by the ImageFormat parameter passed in the `InitVideoCapture(…)` function.

Third, if you simply want to acquire the captured image and do not want to use it for optical marker tracking, then do not add the IVideoCapture implementation to the Scene class.

If you do not have a physical web camera, or want to capture video streaming from a file stored in your local drive (e.g., for overlaying existing video for regression testing), then you can use an application such as VCam^{TM} 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.

4.2 Optical marker tracking using ARTag

After at least one video capture device is added to the Scene class after initialization, you should instantiate one IMarkerTracker (GoblinXNA.Device.Vision.Marker.IMarkerTracker) implementation. Goblin XNA includes a marker tracker class using the ARTag library. (At least one additional optical tracking library will be supported in a future release.) After you instantiate one of the IMarkerTracker implementations, you will need to initialize the tracker by calling `InitTracker(…)` with appropriate parameters.

The first and second parameters are the focal point of the video capture device. This will differ from one camera to another, so you should measure it for the camera you are using for best tracking results. The Graphics & Media Lab at MSU (Moscow State University) provides a great tool for camera calibration, which you can download for free from http://research.graphicon.ru/calibration/gml-matlab-camera-calibration-toolbox.html. The third parameter is a list of settings that define the marker configuration specific to the marker tracker library you use. For ARTag, it would be the marker configuration file that defines the coordinates of the marker arrays and single markers. Please refer to the ARTag documentation for details on how to define those coordinates. 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 Section 2.3.6 on 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 set the `IMarkerTracker.StaticImageFile` property. In this case, you do not need to add any video capture devices to the Scene class. However, if you want to display the static image as the background, you will still need to set `Scene.ShowCameraImage` to true.

4.3 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 use for marker tracking by changing `Scene.TrackerVideoID`. If the video capture devices have different focal points, then you should set the `IVideoCapture.FocalPoint` property for each video capture device you add. Otherwise, the marker tracker library will use the same focal point no matter which capture device you select. Note that significant overhead will be incurred if you switch frequently among different capture devices that have different focal points or resolutions, because ARTag needs to be reinitialized each time the focal point, resolution, or image type (e.g,, color or grayscale) changes. Therefore, if you plan to switch among different capture devices frequently, it is best if they are all the same.

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 video device ID when you add an IVideoCapture implementation to the Scene, and the video device ID added last is used by default.

4.4 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 video ID.

Goblin XNA also lets you specify how many pixels you want the video images to be shifted for each of the left and right eyes by setting `Scene.LeftEyeVideoImageShift` and `Scene.RightEyeVideoImageShift` properties. Shifting the images horizontally toward or away from each other corresponds to moving the point at which your eyes are looking closer or further away from the viewer. The specific setting that will be "right" will depend on the application and the display hardware If you do not set `LeftEyeVideoID` and `RightEyeVideoID` while the virtual scene is rendered in stereo, then the same video (camera) image will be shown to both eyes without shifting.

5 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. 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.

5.1 Physics engine interface

We define an interface, IPhysics, for a 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 a default physics engine implementation that wraps the existing Newton Game Dynamics 1.53 library.

5.2 Physical properties

Before you can add an object that either implements or contains the IPhysicsObject interface (`GeometryNode.Physics` property contained in a Geometry node, but not limited to a Geometry node as long as it either implements or contains the IPhysicsObject interface), 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.

5.3 Physical 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. The material name of a physics object can be set in the IPhysicsObject interface (e.g., in a Geometry node, with the `Geometry-`

`Node.Physics.MaterialName` property). For 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 an IPhysicsMaterial 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 an IPhysicsMaterial object.

These are the material properties:[1]

- 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.
    - 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 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 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.

[1] These descriptions are specific to NewtonPhysics, and they can be different if another IPhysics implementation is used.

5.4 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.
- Specify physical material properties, such as elasticity, static and kinetic friction, and "softness" (IPhysicsMaterial).
- 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 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 Section 5.2 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 library.

Since the notions 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:[2]

[2] These meanings are specific to the NewtonPhysics implementation, and they can be different for different implementation of the IPhysics interface.

- 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 the object is added to the Newton 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.
- `Interactable` is set to true, but `collidable` is set to false.
    - 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, but 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 mostof the capabilities of the original Newton 1.53 library in NewtonPhysics. To perform more advanced simulation, you can directly access the full functionality of the original Newton 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 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.

6 User Interface

Goblin XNA supports a set of common 2D graphical user interface (GUI) components that can be overlaid on the scene.

> 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.

6.1 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.

6.1.1 Base 2D GUI component

All 2D GUIs 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.

6.1.2 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 is 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.

6.1.3 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.

6.1.4 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 listener by calling the `G2DButton.AddActionListener(…)` function, and the listener 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.

6.1.5 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 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.

6.1.6 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.

6.1.7 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` to true, respectively. You will also need to set `G2DSlider.MajorTickSpacing` and `G2DSlider.MinorTickSpacing` to see the major tick marks and minor tick marks, respectively. 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 listener for G2DSlider is ChangeListener, and you can add an implementation of the listener by calling `G2DSlider.AddChangeListener(…)`. Then, the implemented `State-Changed(…)` function will get called whenever the value of the slider changes.

6.1.8 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 listener for G2DTextField is CaretListener, and you can add an implementation of the listener by calling `G2DTextField.AddCaretListener(…)`. Then, the implemented `CaretUpdate(CaretEvent evt)` function will get called whenever the caret position changes. The CaretEvent parameter contains the current position of the caret.

6.1.9 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.

6.2 Event handling

Like Java.Swing, Goblin XNA handles events using listeners. There are different types of listeners, and you need to implement the right listener for a specific event. Then, you can register listeners to a G2D component, and when the event occurs, the registered listeners will get called.

For example, you can add a class that implements ActionListener to G2DButton by calling `G2DButton.AddActionListener(…)`. Then, whenever the button is pressed, the implemented `ActionPerformed(…)` function will get called. The ActionEvent parameter contains the source class that called this `ActionPerformed(…)` class.

6.3 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.

7 Sound

We created a wrapper on top of 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 3.0. To learn how to compile an XACT project file, see http://msdn2.microsoft.com/en-us/library/bb172335(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/bb172307(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.

8 Shader

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.

8.1 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 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.

8.2 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.

8.3 Other shaders

In addition to the 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, 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 */data/Shaders* directory to your project content.

ShadowMapShader is 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.CastShadows` and/or `Model.ReceiveShadows` to true for each 3D model you want to cast or receive shadows. (See **Tutorial 8** for an example.) Note that there are several known problems with the current version of ShadowMapShader. For example, sometimes the shadow is not correctly cast from a source object onto a destination object if the destination object is too close to the source object. In order to use shadow mapping, you will need to add the *PostScreenShadowBlur.fx* and *ShadowMap.fx* files located in the */data/Shaders* and the *ShadowDistanceFadeoutMap.dds* file located in the */data/Textures* directories to your project content.

9 Networking

XNA 2.0 and above 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. 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. Then you need to specify whether the application is a server or a client by setting `State.IsServer` to true or false. Now, 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: LidgrenServer and LidgrenClient.

You will need to assign the IServer or IClient implementation to `Scene.NetworkServer` or `Scene.NetworkClient`, respectively. When you assign them, the Scene class will automatically call the `IServer.Initialize()` function or `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. 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.

9.1 Server

A server implementation should implement the Goblin XNA IServer interface. Even though you can manually call the broadcast or receive functions to communicate between the clients, Goblin XNA automates this process through the INetworkObject interface. (See Section 9.3 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.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.

9.2 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 Section 9.3 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.

9.3 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.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, the 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, which is automatically called by your Game class's `Update(…)` function in `base.Update(…)`. 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 the messages are transferred as arrays of bytes, and our GoblinXNA.Helpers.ByteHelper class provides several utility functions, such as concatenating bytes and converting bytes to or from other primitive types. (The BitConverter class also provides many conversion functions.)

10 Debugging

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

10.1 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 `State.ShowTriangleCount` to true.

10.2 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 Section 11.1.) 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.

10.3 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 check unexpected model behavior.

In order 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.

In order 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.)

11 Miscellaneous

11.1 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 a 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 (e.g., "<var name="SceneFile" value="scene.xml">"), and 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.

11.2 2D shape drawing

Goblin XNA provides several functions for drawing simple 2D shapes such as line, rectangle, and circle. Similar to Java2D, there are draw functions and fill functions. These functions can be found in the GoblinXNA.UI.GUI2D.UI2DRenderer class.

11.3 Performance

Goblin XNA takes advantage of multi-core CPU machine by multi-threading certain operations to speed up the rendering. However, if your machine has a single-core CPU, using multi-threading may result in noticeably worse performance than not using multi-threading. In this case, you may want to set `State.IsMultiCore` to false. This property is set to true by default.

In most of the tutorials, `Scene.PreferPerPixelLighting` is set to true. This property is used only by the SimpleEffectShader. As noted in the tutorials, if your machine has an integrated graphics accelerator (many of which rely extensively on the CPU for certain graphics operations) instead of a discrete graphics accelerator, setting this to true may severely reduce performance. (If you have any doubts about your configuration, we suggest that you compare performance on both settings and choose the faster one!)