IxGame 3D

Content

1. Game class

The IxGame::Game class is the base class for all your games created with the IxGame framework. You are required to extend this class using C++ and to override the core game and lifecycle event methods initialize, finalize, update, and render. This is where you?ll write your code to load the game assets and apply game logic and rendering code. Under the hood, the game class will receive events and act as an abstraction between the running game and the underlying platform layer that is running the game loop and reacting to operating systems.

There are four methods you must implement to get started in writing your own game:

```
#include "IxGame.h"
using namespace IxGame;
class MyGame : public Game
{
	void initialize();
	void finalize();
	void update(float elapsedTime);
	void render(float elapsedTime);
};
```

The Game::initialize() and Game::finalize() methods are called when the game starts up and shuts down, respectively. They are the methods to which you'll add code to load your game assets and cleanup when the game has ended. The Game::update() and Game::render() methods are called once per frame from the game loop implemented in the IxGame::Platform for each operating system. This allows you to separate the code between handling updates to your game's state and rendering your game's visuals. You can use a variety of built-in classes to help with the game rendering.

1.1 Accessing the game instance

The IxGame::Game class can be accessed from anywhere in your game code. It implements a singleton design pattern. Call the static method Game::getInstance() to gain access to the instance of your game from any code.

1.2 Graphics and audio devices

After your game has started, the underlying graphics and audio devices will automatically initialize. This happens prior to the Game::initialize() method being called and readies any classes that use OpenGL or OpenAL. The graphics devices of your Game will be set up with a default 32-bit color frame buffer, a 24-bit depth buffer, and an 8-bit stencil buffer ready for your use. These are the active graphics hardware buffers, which are rendered into from your rendering code.

For more advanced usage, you can apply alternative frame buffers using the IxGame::FrameBuffer class. Immediately after the Game::render() method, the frame buffer is swapped/presented to the physical display for the user to see. You can invoke the Game::clear() method to clear the buffers through any of the methods. You can also call Game::renderOnce() from code, such as from the Game::initialize() method, to callback onto a method that will be called only once and then swapped/presented to the display. This is useful for presenting ad-hoc updates to the screen during initialization for rendering, such as loading screens.

1.3 Game sub-system controllers

The IxGame::Game class also manages game sub-system controllers, such as audio, animation and physics controllers, and provides access to them directly using getter methods. These classes act as controlling interfaces for managing and playing audio and animations that are active in the game, as well as updates to dynamics in the physics systems. These controllers are hosted by the IxGame::Game class and react on lifecycle events being handled in the game.

1.4 Game time and state

Once the instance of a IxGame::Game class has started, the game starts a running time. You can call the Game::getGameTime() to determine how long a game has been running. You can also call Game::getAbsoluteTime() to determine the absolute time that has elapsed since the first Game::run() call. This includes any paused time too. You can call the Game::pause() method and the game will be put into the Game::PAUSED state. If the user on the platform puts the game into the background, the game time is also paused. If the user puts the game back into the foreground, the game will invoke Game::play() and the game will resume. At any time in the game you can determine the game state by calling Game::getState(). The game state can be UNINITIALIZED, RUNNING or PAUSED.

2. Input and Sensors

By creating your game and extending IxGame::Game, you'll be able to add all the required handlers of input events. Additionally, there are methods on IxGame::Game to poll for the current sensor data. This architecture insulates you, as a developer, from the platform-specific details on handling keyboard, touch and mouse events, and from polling the accelerometer state. The following illustrates overridden methods to handle input events:

```
#include "IxGame.h"
using namespace IxGame;
class MyGame : public Game
{
    // Input events
    void keyEvent(Keyboard::KeyEvent evt, int key);
    void touchEvent(Touch::TouchEvent evt, int x, int y, unsigned int contactIndex);
    bool mouseEvent(Mouse::MouseEvent evt, int x, int y, int wheelDelta);
    void gamepadEvent(Gamepad::GamepadEvent evt, Gamepad* gamepad);

    // Input methods
    void displayKeyboard(bool display);
    void registerGesture(Gesture::GestureEvent evt);
```

```
    void getAccelerometerValues(float* pitch, float* roll);
    // ...
};
```

2.1 Mouse

Game::mouseEvent() is called when a mouse event occurs.
Mouse events that are not consumed will be interpreted as a touch event. You can consume a mouse event by overridding Game::mouseEvent() and returning true. This gives you the option to uniquely handle mouse events from touch events. Game::mouseEvent() returns false by default.
Note that some mobile devices can use a Bluetooth mouse.

Mouse Capture
Game::setMouseCaptured() can be used to enable mouse capture. While mouse capture is enabled, all mouse move events will then be delivered as deltas instead of absolute positions.
Game::setCursorVisible() can be used to hide the mouse cursor.

2.2 Touch

Game::touchEvent() is called when a touch event occurs. x and y are screen coordinates where the top left is 0,0. Some platforms may allow you to touch outside the bounds of the screen so negative x and y values are possible.

Multi-touch
You can enable multi-touch using Game::setMultiTouch(). The Game::touchEvent() parameter contactIndex is used to differentiate the touch contacts. Do not assume that the contactIndex values are sequential.

Keyboard
Game::keyEvent() is called when a keyboard event occurs.

Showing the virtual keyboard
You can call Game::displayKeyboard() to show or hide a virtual keyboard for platforms that support it.

2.3 Gestures

Some platforms support gestures. Game::isGestureSupported() can be used to determine which gestures are supported.

Game::registerGesture() is used to register for a type of gesture. Once a gesture is registered, you will receive callbacks via Game::gestureSwipeEvent(), Game::gesturePinchEvent() or Game::gestureTapEvent().

2.4 Accelerometer

You can call Game::getAccelerometerValues() and pass in pointers to parameters that will be populated with the current sensor values for the accelerometer.

Despite its name, implementations of Game::getAccelerometerValues() on the various platforms are at liberty to utilize data from other sensors as well. Gyro data may be incorporated in fusion with accelerometer data to improve transient response for a better indication of device orientation. Therefore Game::getAccelerometerValues() is to be used as an indicator of device orientation rather than strictly net acceleration. For games that need to distinguish between acceleration and device rotation rate, the current plan is to introduce separate functions for each.

2.5 Gamepad

You can listen for gamepad events via Game::gamepadEvent(Gamepad::GamepadEvent evt, Gamepad* gamepad). This will return you a pointer to a IxGame::Gamepad instance that has been connected or disconnected. You can then poll the gamepad's state each frame within your game's update() method.

You can also use a game.config file to define one or more virtual gamepads from a UI form. These virtual gamepads are typically used on mobile devices when a gamepad is not connected -- instead, a touch screen is used to interact with the form. The form contains joystick and button controls with data bindings to the appropriate button mappings defined in the Gamepad class.

Ex. within game.config:

```
gamepad playerOne
{
    form = res/common/gamepadP1.form
}

gamepad playerTwo
{
    form = res/common/gamepadP2.form
}
```

This will create two gamepads at startup. A connected event will be triggered for each virtual gamepad when it is created. Both the Character and Racer samples make use of gamepad events, using virtual gamepads if no physical device is available, so you can take a look at their source code to see how we're currently handling input.

3. Model and Mesh

The IxGame::Model class is the basic component used to draw geometry in your scene. The model contains a few key elements: a IxGame::Mesh, an optional IxGame::MeshSkin and one or more IxGame::Material. These contribute to the information that is needed to perform the rendering of a model.

3.1 Mesh geometry

The IxGame::Mesh class consists of a IxGame::VertexFormat attribute. This attribute describes the layout for the vertex data as well as the actual vertex data, which is used as input in the rendering of the geometry. In addition, it holds one or more IxGame::MeshParts. These parts define the primitive shapes and indices into the vertex data that describe how the vertices are connected.

Game artists use 3-D modeling tools that are capable of organizing and splitting the vertex data into parts/subsets based on the materials that are applied to them. The IxGame::Mesh class maintains one vertex buffer to hold all the vertices, and for each IxGame::MeshPart, an index buffer is used to draw the primitive shapes.

3.2 MeshSkin and Joints

The IxGame::Mesh class supports an optional IxGame::MeshSkin. This is used when loading models that represent characters in the game that have a skeleton consisting of IxGame::Joint objects (bones). Vertex skinning is the term used to describe the process of applying a weighting or relationship with the Joints and their affected vertices. Using 3-D modeling tools, artists can add this additional weighting information onto the vertices in order to control how much a particular vertex should be impacted. This is based on the transformation of joints that can affect them. You will learn later how to apply special, skinned Materials that support this weighting. The IxGame 3-D framework supports a maximum of four blend weights per vertex. The IxGame::MeshSkin class holds and maintains a hierarchy of IxGame::Joint objects that can be transformed. A typical operation is to animate the transformation (usually only rotations) of the joints. The data within this class can be bound onto skinned Materials to ensure proper impact of weights onto their influenced vertices.

4. Lights

The IxGame::Light class can be attached to any IxGame::Node in order to add lighting information into a IxGame::Scene. This lighting information must be bound to the IxGame::Material that is being applied onto the IxGame::MeshParts. There are three types of lights in the IxGame 3-D framework - directional, point, and spot lights.

All IxGame::Light components can be loaded into a IxGame::Scene using the IxGame::Bundle class. However, it is your responsibility to bind the relevant lighting information stored in the light into the IxGame::Material class.

You can also programmatically create a light using the factory methods on the IxGame::Light class. Here is an example of how to create and add a directional light to your scene and bind the lighting information onto a model's material(s):

```
void MyGame::initialize()
{
    // ...
    // Create a node and light attaching the light to the node
```

```
    Node* lightNode = Node::create("directionalLight1");
    Light* light = Light::createDirectional(Vector3(1, 0, 0));
    lightNode->setLight(light);

    // Bind the relevant lighting information into the materials
    Material* material = _modelNode->getModel()->getMaterial();
    MaterialParameter* parameter = material->getParameter("u_lightDirection");
    parameter->bindValue(lightNode, &Node::getForwardVectorView);
}
```

4.1 Pre-computed lighting maps

Adding lighting information into IxGame::Material adds computationally expensive graphics computations. In many games, there are usually multiple static lights and objects in the scene. In this relationship, the additive light colors contributing to the objects can be pre-computed during the design phase. 3-D modeling tools typically support the ability to compute the light's additive color contributions using a process called baking. This process allows the artist to direct the contributing light and color information into a separate or combined texture so that this is not required during the rendering.

You can optionally declare and pass in pre-generated light maps using the colored-unlit.frag /textured-unlit.frag shaders and specifying in your materials defines = TEXTURE_LIGHTMAP. Then you just assign them using the sampler m_lightmapTexture in your material definition to the image that was pre-generated that contains the light+color for your object. It is recommended to use 8-bit alpha textures to reduce the size.

4.2 Directional lights

In most games, you'll want to add a IxGame::Light class whose type is Light::DIRECTIONAL. This type of light is used as the primary light source, such as a sun or moon. The directional light represents a light source whose color is affected only by the constant direction vector. It is typical to bind this onto the IxGame::Materials of objects that are dynamic or moving.

4.3 Point and spot lights

Due to the expensive processing overhead in using point and spot lights, many games are designed to restrict point and spot light use to be static, baked into light and color maps. However, the point and spot light types add exceptional realism to games. Using them in separate or combined rendering passes, you can bind point and spot lights into material to add dynamic point and spot light rendering. All the built-in IxGame .materials files support directional, point and spot lights. Also, with minor modification to the shaders, you can add additional passes to combine two or more lights. It should be noted that there is a significant performance impact in doing this. For these cases, you'll usually want to restrict the influence of lights on a material to no more than the one or two closest lights at a time. This can be achieved by using a simple test in the Game::update() method to find the closest light to a IxGame::Model and then bind them to the IxGame::Material once they are found.

5. Materials and Shaders

The IxGame 3-D framework uses a modern GPU shader based rendering architecture and uses OpenGL 2.0+ (desktop) or OpenGL ES 2.0 (mobile) along with the OpenGL Shading Language (GLSL). Currently, all the code in graphics-related classes uses the OpenGL hardware device directly.

5.1 Using materials

The IxGame::Material class is the high level definition of all the rendering information needed to draw a IxGame::MeshPart. When you draw a IxGame::Model, the mesh's vertex buffer is applied and for each IxGame::MeshPart its index buffer(s) and IxGame::Materials are applied just before the primitives are drawn.

5.2 RenderState and Effects

Each IxGame::Material consists of a IxGame::RenderState and a IxGame::Effect. The IxGame::RenderState stores the GPU render state blocks that are to be applied, as well as any IxGame::MaterialParameters to be applied to the IxGame::Effect. While a IxGame::Material is typically used once per IxGame::MeshPart, the IxGame::Effect is created internally based on the unique combination of selected vertex and fragment shader programs. The IxGame::Effect represents a common reusable shader program.

5.3 Techniques

Since you can bind only one IxGame::Material per IxGame::MeshPart, an additional feature is supported that's designed to make it quick and easy to change the way you render the parts at runtime. You can define multiple techniques by giving them different names. Each one can have a completely different rendering technique, and you can even change the technique being applied at runtime by using Material::setTechnique(const char* name). When a material is loaded, all the techniques are loaded ahead too. This is a practical way of handling different light combinations or having lower-quality rendering techniques, such as disabling bump mapping, when the object being rendered is far away from the camera.

5.4 Creating materials

You can create a IxGame::Material from the simple IxGame::Properties based .material files. Using this simple file format, you can define your material, specifying all the rendering techniques and pass information.

Here is an example of loading a .material file:

```
Material* planeMaterial = planeNode->getModel()->setMaterial("res/floor.material");
```

Setting vs. binding material parameters

Once you have created a IxGame::Material instance, you'll want to get its parameters and then set or bind various values to them. To set a value, get the IxGame::MaterialParameter and then call the appropriate setValue() method on it. Setting material parameter values is most common in parameters that are based on values that are constants.

Here is an example of setting a value on a parameter:

```
material->getParameter("u_diffuseColor")->setValue(Vector4(0.53544f, 0.53544f, 0.53544f, 1.0f));
```

For values that are not constants and are determined from other objects, you'll want to bind a value to it. When binding a value, you are giving the parameter a function pointer that will only be resolved just prior to rendering. In this example, we will bind the forward vector for a node (in view space).

Here is an example of binding a value on a parameter:

```
material->getParameter("u_lightDirection")->bindValue(lightNode, &Node::getForwardVectorView);
```

.material files
As you can see in the following .material file, we have one Material, one Technique and one Pass. The main parts of this material definition are the shaders, uniforms, samplers and renderState. You will see certain upper case values throughout the file. These represent constant enumeration values and can usually be found in the IxGame::RenderState or IxGame::Texture class definitions:

```
material duck
{
    technique
    {
        pass 0
        {
            // shaders
            vertexShader = res/shaders/textured.vert
            fragmentShader = res/shaders/textured.frag
            defines = SPECULAR

            // uniforms
            u_worldViewProjectionMatrix = WORLD_VIEW_PROJECTION_MATRIX
            u_inverseTransposeWorldViewMatrix                                    =
            INVERSE_TRANSPOSE_WORLD_VIEW_MATRIX
            u_cameraPosition = CAMERA_WORLD_POSITION

            // samplers
            sampler u_diffuseTexture
            {
                path = res/duck-diffuse.png
                mipmap = true
                wrapS = CLAMP
                wrapT = CLAMP
                minFilter = NEAREST_MIPMAP_LINEAR
                magFilter = LINEAR
```

```
            }
            // render state
            renderState
            {
                cullFace = true
                depthTest = true
            }
        }
    }
}
```

5.5 Built-in shaders

The <IxGame-root>/IxGame/res/shaders directory contains a set of the most common shaders used in your games. To reduce shader code duplication the IxGame framework also supports declaring including of shader files within vertex and shader program files.

If there is an error compiling the shaders the expanded shader without the definitions is output with an .err file extension in the same directory where the file was loaded from.

Example:

#include "lib/lighting.frag"

Shader preprocessor definitions

Using pre-processor definitions, the built-in shaders support various features. Adding certain shader definitions (defines=XXX) will require use specific uniform/samplers 'u_xxxxxxx'. You must set these in your vertex stream in your mesh and/or material parameters.

Property inheritance
When making materials with multiple techniques or passes, you can put any common things, such as renderState or shaders, above the material or technique definitions. The IxGame::Property file format for the .material files supports property inheritance. Therefore, if you put the renderState in the material sections, then all techniques and passes will inherit its definition.

6. Scene and Nodes

At the heart of any game is a visual scene. Using the IxGame::Scene class, you can create and retain a rich 3-D scene for organizing visual, audio, animation and physics components in your game.

The IxGame::Scene class is based on a hierarchical data structure that is often referred to as a scene graph. Using the IxGame::Scene and IxGame::Node classes, you can build up a game level by attaching various game components to the nodes in the scene. The node will maintain the transformation for any attachments. As a basic example, a scene might have two nodes. The first node could have a IxGame::Camera attached to it and the second node could have a IxGame::Model attached to it. The IxGame::Scene will have the camera set as the active camera. You could then transform either/both of the nodes to change the player's perspective

on what they will see in the game.

There are a variety of components you can attach to the IxGame::Node class:

Component	Description
IxGame::Model	Used to represent the mesh/geometry in the scene.
IxGame::Camera	Used to represent a view/perspective into the scene.
IxGame::Light	Used to hold lighting information that can affect how a Model is rendered.
IxGame::PhysicsCollisionObject	Used to define the basic physics dynamics that will be simulated.
IxGame::ParticleEmitter	Used to represent smoke, steam, fire and other atmospheric effects.
IxGame::Terrain	Used to represent a height map based terrain.
IxGame::AudioSource	Used to represent a source where audio is being played from.
IxGame::Form	Used to represent a user interface that can be in a scene.

A typical flow will have you loading/building a large scene representing all the components needed in the game level. This is done once during Game::initialize(). For every call to the Game::update() method, the code will update changes to the nodes and attached components based on events such as user input . Then the application will traverse the scene and render the parts in the scene that are visible from scene's active camera.

Exporting a 3-D scene from Autodesk Maya/Max
If you want to export 3-D scenes, use the native FBX Export (for FBX).

Exporting a 3-D scene from Blender
Blender supports exporting to FBX file format.

File > Export > Autodesk FBX (.fbx)

6.1 Binary encoding a scene

Run IxGame-encoder with no arguments to see the usage information and supported arguments.

Usage: IxGame-encoder [options] <filepath>
Example

Convert the FBX file duck.fbx into IxGame binary file duck.gpb.

IxGame-encoder duck.fbx

6.2 Encoding an FBX file

To convert an FBX file to a IxGame binary file, you must install the FBX SDK and set the preprocessor directive USE_FBX. See the instructions in the IxGame-encoder README.

6.3 Loading a scene

Using the IxGame::Bundle class, you can load either an entire scene or various parts of a scene into any existing scene. The IxGame::Bundle parses the binary file and de-serializes the objects from the file so that you can use them in your game.

Here is an example of loading a simple scene containing a model of a duck, a light, and a camera from a IxGame binary file:

```
void MeshGame::initialize()
{
	// Load the scene from our IxGame binary file
	Bundle* bundle = Bundle::create("res/duck.gpb");
	Scene* scene = bundle->loadScene();
	SAFE_RELEASE(bundle);

	// Get handles to the nodes of interest in the scene
	_modelNode = scene->findNode("duck");
	Node* _lightNode = scene->findNode("directionalLight1");
	Node* _cameraNode = scene->findNode("camera1");

	// More initialization ...
}
```

6.4 Updating a scene

After handling input events or polling the sensors, it's time to update the scene. It is very important to understand the scene representing your game level. We always want to update things that are impacted by the changes to optimize performance. In order to optimize the performance of your game, it is essential that you only update objects that need to be changed. In this example, we'll apply a rotation when the user has touched the screen or mouse button:

```
void MyGame::update(float elapsedTime)
{
	// Rotate the model
	if (!_touched)
		_modelNode->rotateY(elapsedTime * MATH_DEG_TO_RAD(0.05f));
	}
}
```

Some examples of typical things you will want to update in your scene may include:

applying forces or impulses onto rigid bodies
applying transformations
starting or stopping animations
showing or hiding components

6.5 Rendering a scene

To render a scene you'll need to gather all the models in the scene that are attached to nodes and then draw them. Calling the Scene::visit() method, the scene's hierarchical data structure is traversed and for each node in the scene, the specified method is invoked as a callback.

```
void MyGame::render(float elapsedTime)
{
	// Clear the buffers to black
	clear(CLEAR_COLOR_DEPTH, Vector4::zero(), 1.0f, 0);
	// Visit all the nodes in the scene, drawing the models/mesh.
	_scene->visit(this, &MyGame::drawScene);
}

bool MyGame::drawScene(Node* node, void* cookie)
{
	// This method is called for each node in the scene.
	Model* model = node->getModel();
	if (model)
		model->draw();
	return true;
}
```

6.6 Culling non-visible models

In some scenes, you may have many models contributing to the game level. However, with a moving camera, only some models will be visible at any particular time. Running the code in the snippet above on much larger scenes would cause many models to be drawn unnecessarily. To avoid this, you can query a IxGame::Node class and retrieve a IxGame::BoundingSphere using Node::getBoundingSphere(). This bound represents an approximation of the representative data contained within a node. It is only intended for visibility testing or first-pass intersection testing. If you have a moving camera with many objects in the scene, ensure that you add code to test visibility from within your visitor callback. This will ensure the node is within the camera's viewing range. To do this, make a simple intersection test between the front of each node and the active camera frustum (by calling Camera::getFrustum()) that represents the outer planes of the camera's viewing area. Here is a snippet of code to perform such an intersection test:

```
bool MeshGame::drawScene(Node* node, void* cookie)
{
	// Only draw visible nodes
	if (node->getBoundingSphere()->intersect(_camera->getFrustum())
	{
		Model* model = node->getModel();
		if (model)
			model->draw();
	}
	return true;
}
```

7. Animation

Animation is a key component to bringing your game to life. Within IxGame, there is support to create both property animations and character animations. The IxGame::Animation class provides factory methods for creating animations on properties of classes that extend IxGame::AnimationTarget. Character animations from within the scene file are imported and stored on the IxGame::AnimationTarget they target. All animations on a IxGame::AnimationTarget can be obtained by ID.

7.1 AnimationTargets

IxGame::Transform, IxGame::Node, and IxGame::MaterialParameter are animation targets.

Animations can be created on the scale, rotation and translation properties of the IxGame::Transform. Animations can also target any IxGame::Node, which extends IxGame::Transform.

Also, animations can target instances of IxGame::MaterialParameter. Any parameters on a material of type float, integer, or 2-, 3-, and 4-dimensional vectors can be animated.

7.2 Creating property animations

Animations are created from the IxGame::AnimationTarget. AnimationTarget provides methods to create simple two key frame animations using createAnimationFromTo(), and createAnimationFromBy(). Multiple key frame sequences can be created from createAnimation().

Here is an example of how to create a multiple key frame animation on a node's translation properties:

```
unsigned int keyCount = 3;
unsigned long keyTimes[] = {0L, 500L, 1000L};
float keyValues[] =
{
      0.0f, -4.0f, 0.0f,
      0.0f, 0.0f, 0.0f,
      0.0f, 4.0f, 0.0f
};
Animation*sampleAnim = enemyNode->createAnimation("sample", Transform::ANIMATE_TRANSLATE,
                                                  keyCount, keyTimes, keyValues,
                                                 Curve::LINEAR);
```

Here is the same animation specified in a .animation file that can also be loaded by the IxGame::AnimationTarget:

```
animation sample
{
      property = ANIMATE_TRANSLATE
```

```
    keyCount = 3
    keyTimes = 0, 500, 1000
    keyValues = 0.0 -4.0 0.0 0.0 0.0 0.0 0.0 4.0 0.0
    curve = LINEAR
}
```

To create the animation from this file you would call the following code:

```
Animation* sampleAnim = enemyNode->createAnimation("sample", "sample.animation");
```

7.3 Curves

There are many different interpolation types defined within the IxGame::Curve class that can be used to interpolate through the animation data.

7.4 Character animations

Character animations are complex because they can be composed of multiple animations targeting numerous joints within a character model. For this reason, character animations are usually included within the scene file and are imported when the .gpb file is loaded. To simplify and optimize all animations under single animation. The IxGame-encoder supports grouping all the animation on joints leading up to a common root joint under a single animation. This is an option in the IxGame-encoder using the -groupAnimations or -g option. This groups them under a single animation called ‘animations’.

These animations can be obtained by calling AnimationTarget::getAnimation() specifying the animation’s ID.

7.5 AnimationClips

A IxGame::AnimationClip is created from the IxGame::Animation class and is a snapshot of the animation that can be played back, manipulated with speed and repeated.

Here is an AnimationClip that has been created from a character animation of an elf:

```
AnimationClip* elfRun = elfAnimation->createClip("elf_run", 200L, 310L);
elfRun->setRepeatCount(AnimationClip::REPEAT_INDEFINITE);
elfRun->setSpeed(2.0f);
```

Animation clips can be specified within an .animation file that can be given to an animation to create clips. The total number of frames that make up the animation must be specified in the file. The begin and end parameters of each clip are specified in frames. An assumption that the animation runs at 60 frames per second has been made. Here is an example of an .animation file for an elf animation:

```
animation elf
{
    frameCount = 350
```

```
    clip idle
    {
        begin =    0
        end = 75
        repeatCount = INDEFINITE
    }
    clip walk
    {
        begin = 75
        end = 200
        repeatCount = INDEFINITE
    }
    clip run
    {
        begin = 200
        end = 310
          repeatCount = INDEFINITE
        speed = 2.0
    }
    clip jump
    {
        begin = 310
        end = 350
        repeatCount = 1
    }
}
```

Animations can be played back by calling Animation::play(), passing a clip ID, or by calling AnimationClip::play() directly on the clip. Animations can also be paused and stopped in the same fashion.

7.6 Animation blending

The IxGame::AnimationClip class has a blend weight property that can be used to blend multiple animations. There is also a method called AnimationClip::crossFade() that conveniently provides the ability to fade the currently playing clip out and fade in the specified clip over a given period of time.

7.7 AnimationClip listeners

Animation events can be triggered on a IxGame::AnimationClip by registering instances of IxGame::AnimationClip::Listener with the clip. The listeners can be registered to be called back at the beginning or end of the clip, or at any specific time throughout the playback of the clip. This can be useful for starting a particle emitter when a character's foot hits the ground in an animation, or to play back a sound of a gun firing during an animation of an enemy shooting.

8. Text and Fonts

The IxGame::Font class is used to draw 2D ASCII text.

Note: Unicode is currently not supported.

##Converting a TrueType font to a "IxGame Bundle" file

You must convert the TrueType font to a "IxGame Bundle File" and include the .gpb file in your project. The .ttf file does not need to be included.

Find or create a TrueType font file (.ttf).

Pass the .ttf file to the IxGame-encoder command line tool and specify the size you want.

IxGame-encoder -s 28 myfont.ttf

The IxGame-encoder will output a .gpb file.

Copy the .gpb file to your game's resource directory.

Note: Pre-compiled versions of IxGame-encoder is available in IxGame\bin after Installing External Dependencies.

Drawing text with a Font

```
void MyGame::initialize()
{
    // Create a font from the gpb file
    _font = Font::create("res/myfont.gpb");
}

void MyGame::render(float elapsedTime)
{
    // Clear the frame buffer
    clear(CLEAR_COLOR_DEPTH, Vector4(0, 0, 0, 1), 1.0f, 0);

    // Draw the text at position 20,20 using red color
    _font->start();
    char text[1024];
    sprintf(text, "FPS:%d", Game::getFrameRate());
    _font->drawText(text, 20, 20, Vector4(1, 0, 0, 1), _font->getSize());
    _font->finish();
}

void MyGame::finalize()
{
    // Use built-in macros to clean up our resources.
    SAFE_RELEASE(_font);
}
```

Batching Glyphs and Words

It is possible to call Font::drawText() multiple times between Font::start() and Font::finish() in order to draw different text in different places with the same font.

Examples
All of the samples use Font::drawText() to render the framerate.

9. Physics

The IxGame framework supports 3-D physics using the game service/controller IxGame::PhysicsController. The IxGame::PhysicsController class maintains a physics world that has gravity, and will simulate the objects you add to it.

The IxGame physics system supports 3-D rigid body dynamics, including collision shapes, constraints, and a physics character class. To simulate objects within the physics world, you need to create a IxGame::PhysicsCollisionObject object representing the geometry, or IxGame::Model. By attaching a collision object to a IxGame::Node, the rigid body will be added to the physics world and the simulation will automatically update the node's transformation.

9.1 PhysicsCollisionObject

PhysicsCollisionObject is the base class that provides an interface for receiving collision events.

You can add collision listeners to a PhysicsCollisionObject or test if the collision object currently collides with another collision object.

There are 3 types of collision objects:

PhysicsRigidBody
PhysicsGhostObject
PhysicsCharacter
PhyiscsVehicle
PhysicsRigidBody

A rigid body is an idealized, infinitely hard, non-deformable solid object. Rigid bodies have mass, shape and other properties that affect forces within the simulation.

A PhysicsRigidBody can be set to be a kinematic rigid body. A kinematic rigid body is an object that is not simulated by the physics system, and instead has its transform driven manually.

A note regarding the mass of a rigid body: In IxGame, a mass value of zero causes the object to be immovable. This value is reserved by the underlying physics engine for making objects static, (albeit admittedly counter-intuitive).

9.2 Create a PhysicsRigidBody

To create a rigid body, first you need to know what kind of shape you want to simulate. The physics system

supports boxes, spheres, meshes, capsules, and terrain height fields. For basic shapes, such as boxes and spheres, you can programmatically create the rigid bodies by calling Node::setCollisionObject() and passing in the desired shape type.

```
PhysicsRigidBody::Parameters params;
params.mass = 10.0f;
node->setCollisionObject(PhysicsCollisionObject::RIGID_BODY,
                                    PhysicsCollisionShape::box(), &params);
```

All other types of rigid bodies must be created using the .scene and .physics property definition files. The .scene file allows you to bind various attachments or properties to nodes, including a rigid body.

For example, to create a mesh rigid body for the node within the scene with ID equal to tree_1:

```
game.scene:
scene
{
    ...
    node tree_1
    {
        ...
        collisionObject = game.physics#tree_mesh
    }
    ...
}

game.physics:

collisionObject tree_mesh
{
    type = RIGID_BODY
    shape = MESH
    mass = 15.0
    ...
}
```

9.3 RigidBody schema

All properties have default values if not defined. See PhysicsRigidBody::Parameters for more information.

```
collisionObject <string>
{
    type                = <RIGID_BODY | GHOST_OBJECT | CHARACTER>
    shape               = <BOX | SPHERE | MESH | HEIGHTFIELD | CAPSULE>
    image               = <string> // only for HEIGHTFIELD
    radius              = <float>
    height              = <float>
    extents             = <float, float, float>
```

```
        center                  = <float, float, float>
        centerAbsolute          = <float, float, float>

        mass                    = <float>
        friction                = <float>
        restitution             = <float>
        linearDamping           = <float>
        angularDamping          = <float>
        kinematic               = <bool>
        anisotropicFriction = <float, float, float>
        gravity                 = <float, float, float>
}
Shapes    Properties
BOX       extents, center, center-absolute
SPHERE radius, center, center-absolute
MESH
HEIGHTFIELD     image
CAPSULE     radius, height, center, center-absolute
```

9.4 PhysicsGhostObject

A ghost object is like a rigid body except that it does not have an effect the simulation. It will not cause forces or react to the other rigid bodies. Ghost objects have a shape but they do not have mass, or any of the properties that affect forces.

Ghost objects are useful for querying the simulation, or detecting collisions without having rigid bodies react to the ghost object. A ghost object could be used to detect if an object entered a volume, such as a soccer ball going into a goal. This use of a ghost object is often called a volumetric trigger. Ghost objects can also detect if they collide with other ghost objects.

Collision objects do not require a model so you could use a ghost object to check if a camera collides with a wall.

9.5 Creating a PhysicsGhostObject

Programmatically:

```
// Create a ghost object with radius 5
node->setCollisionObject(PhysicsCollisionObject::GHOST_OBJECT,
                                    PhysicsCollisionShape::sphere(5.0f));
```

In a .physics file:

```
collisionObject powerup
{
        type = GHOST_OBJECT
        shape = SPHERE
```

```
    radius = 5.0
}
```

PhysicsGhostObject schema

Ghost objects only have a shape and support the same shapes as rigid bodies.

```
collisionObject <string>
{
    type                = GHOST_OBJECT
    shape               = <BOX | SPHERE | MESH | HEIGHTFIELD | CAPSULE>
    radius              = <float>
    height              = <float>
    extents             = <float, float, float>
    center              = <float, float, float>
    centerAbsolute      = <float, float, float>
    image               = <string>    // HEIGHTFIELD shapes only.
}
```

9.6 PhysicsCharacter

The PhysicsCharacter class can be used to control the movements and collisions of a character in a game. It interacts with the physics system to apply gravity and handle collisions, however dynamics are not applied to the character directly by the physics system. Instead, the character's movement is controlled directly by the PhysicsCharacter class. This results in a more responsive and typical game character than would be possible if trying to move a character by applying physical simulation with forces.

Creating a PhysicsCharacter

To programmatically create a PhysicsCharacter with mass 20 and capsule shape:

```
PhysicsRigidBody::Parameters params(20.0f);
node->setCollisionObject(PhysicsCollisionObject::CHARACTER,
                         PhysicsCollisionShape::capsule(1.2f, 5.0f, Vector3(0, 2.5, 0), true),
&params);
PhysicsCharacter* character = static_cast<PhysicsCharacter*>(
                                                    node->getCollisionObject());
```

PhysicsCharacter schema

Physics characters have a mass and shape. A capsule is a typical shape for a typical biped character.

```
collisionObject <string>
{
    type                = CHARACTER
    shape               = <BOX | SPHERE | MESH | CAPSULE>
    radius              = <float>
    height              = <float>
    extents             = <float, float, float>
    center              = <float, float, float>
    centerAbsolute      = <float, float, float>
```

```
    mass                        = <float>
}
```

9.7 Creating a PhysicsContraint

The IxGame framework supports various types of constraints between two rigid bodies (or one rigid body and the physics world), including hinge, fixed, generic (six-degree-of-freedom), socket, and spring. Constraints can be created programmatically using one of the create functions on IxGame::PhysicsController, or they can be specified within the physics section of the .scene file. For example, to create a hinge constraint from within a .scene file between the rigid body attached to the node with id door and the physics world:

game.scene:

```
scene
{
    ...
    physics
    {
        ...
        constraint
          {
            type = HINGE
            rigidBodyA = door
            rotationOffsetA = 0.0, 1.0, 0.0, 90.0
            translationOffsetA = 0.0, 0.0, 2.0
            limits = 0.0, 90.0, 0.5
        }
    }
}
```

PhysicsConstraint schema

```
constraint <string>
{
    type                    = <FIXED | GENERIC | HINGE |    SOCKET | SPRING>
    rigidBodyA              = <string>
    rigidBodyB              = <string>
    translationOffsetA = <float, float, float>
    translationOffsetB = <float, float, float>
    rotationOffsetA     = <float>
    rotationOffsetB     = <float>
    angularLowerLimit   = <float, float, float>
    angularUpperLimit   = <float, float, float>
    linearLowerLimit    = <float, float, float>
    linearUpperLimit    = <float, float, float>
    limits                  = <float, float, float>
    angularDampingX     = <float>
    angularDampingY     = <float>
    angularDampingZ     = <float>
```

```
    angularStrengthX     = <float>
    angularStrengthY     = <float>
    angularStrengthZ     = <float>
    linearDampingX       = <float>
    linearDampingY       = <float>
    linearDampingZ       = <float>
    linearStrengthX      = <float>
    linearStrengthY      = <float>
    linearStrengthZ      = <float>
    breakingImpulse      = <float>
}
```

9.8 Constraint Types Properties

FIXED

GENERIC translationOffsetA, translationOffsetB, rotationOffsetA, rotationOffsetB, angularLowerLimit, angularUpperLimit, linearLowerLimit, linearUpperLimit

HINGE translationOffsetA, translationOffsetB, rotationOffsetA, rotationOffsetB, limits

SOCKET translationOffsetA, translationOffsetB

SPRING translationOffsetA, translationOffsetB, rotationOffsetA, rotationOffsetB, angularLowerLimit, angularUpperLimit, linearLowerLimit, linearUpperLimit, angularDampingX, angularDampingY, angularDampingZ, angularStrengthX, angularStrengthY, angularStrengthZ, linearDampingX, linearDampingY, linearDampingZ, linearStrengthX, linearStrengthY, linearStrengthZ

9.9 Handling collisions

The IxGame framework allows you to register to be notified whenever a collision occurs between two rigid bodies (and also when two rigid bodies stop colliding). In order to do this, you must define a class that derives from IxGame::PhysicsRigidBody::Listener and implements the function collisionEvent(). Then, you must add an instance of the class as a listener on a given rigid body using the PhysicsRigidBody::addCollisionListener function. For example, to print all information for all collisions with the door and for collisions between the character and the wall:

MyGame.h

```
class MyGame: public IxGame::PhysicsRigidBody::Listener
{
public:
    /// ...
    /**
     * Collision event handler.
     */
    void collisionEvent(PhysicsRigidBody::Listener::EventType type,
                        const PhysicsRigidBody::CollisionPair& pair,
    const Vector3& pointA, const Vector3& pointB);
    // ...
};
```

MyGame.cpp:

```
MyGame* mygame;
Node* door;
Node* character;
Node* wall;
// ...
door->getRigidBody()->addCollisionListener(mygame);
character->getRigidBody()->addCollisionListener(mygame, wall);
// ...
void MyGame::collisionEvent(PhysicsRigidBody::Listener::EventType type,
                            const PhysicsRigidBody::CollisionPair& pair,
                            const Vector3& pointA, const Vector3& pointB)
{
    GP_WARN("Collision between rigid bodies %s (at point (%f, %f, %f)) "
            "and %s (at point (%f, %f, %f)).",
            pair._rbA-getNode()-getId(), pointA.x, pointA.y, pointA.z,
            pair._rbB-getNode()-getId(), pointB.x, pointB.y, pointB.z);
}
```

9.10 PhysicsVehicle

The PhysicsVehicle and PhysicsVehicleWheel classes give you access to vehicle physics for racing games. You designate a VEHICLE type collision object for the node in your scene representing the vehicle body or chassis, and VEHICLE_WHEEL for the nodes representing the wheels. When IxGame loads the scene, it attempts to automatically bind the wheels to the associated vehicle chassis. In the hierarchy of your scene be sure to locate the wheel nodes and the vehicle chassis under a common group node. This is how IxGame determines which vehicle body the wheels belong to – by searching for a common ancestor in the hierarchy. The nodes do not need to be direct descendents of the group node; they just need to appear somewhere below it in the hierarchy. The presence of other mesh nodes under the common node has no effect. All that matters is that the node of collision type VEHICLE shares a common ancestor with the nodes of collision type VEHICLE_WHEEL (details below):

Car model in Maya

Then in your .scene file, designate collision objects for the car and its wheels, like this:

```
scene main
{
    path = res/common/game.gpb
    activeCamera = camera1
    node carbody
    {
        url = car_top
        material = res/common/game.material#car
        collisionObject = res/common/game.physics#car
    }
    node wheelFrontLeft
```

```
    {
        url = Left_top
        material = res/common/game.material#car
        collisionObject = res/common/game.physics#carWheelFrontLeft
    }
    node wheelFrontRight
    {
        url = Right_top
        material = res/common/game.material#car
        collisionObject = res/common/game.physics#carWheelFrontRight
    }
    node wheelBackLeft
    {
        url = Left_bottom
        material = res/common/game.material#car
        collisionObject = res/common/game.physics#carWheelBackLeft
    }
    node wheelBackRight
    {
        url = Right_bottom
        material = res/common/game.material#car
        collisionObject = res/common/game.physics#carWheelBackRight
    }
//...
    physics
    {
        gravity = 0.0, -9.8, 0.0
    }
}
```

In the .physics file be sure to specify type VEHICLE for the chassis and type VEHICLE_WHEEL for the wheels:

```
collisionObject car
{
    type = VEHICLE

    shape = BOX
    mass = 800.0
    friction = 0.5
    restitution = 0.01
    linearDamping = 0.025
    angularDamping = 0.6

    steeringGain = 0.4
    brakingForce = 350.0
    drivingForce = 2000.0

    steerdownSpeed = 87
    steerdownGain = 0.22
```

```
    brakedownStart = 100
    brakedownFull = 170
    drivedownStart = 105
    drivedownFull = 180
    boostSpeed = 74
    boostGain = 2.6
    downforce = 4.5
}

collisionObject carWheel
{
    type = VEHICLE_WHEEL

    shape = MESH
    mass = 1.0
    friction = 0.5
    restitution = 0.01
    linearDamping = 0.025
    angularDamping = 0.16

    wheelDirection = 0, -1, 0
    wheelAxle = -1, 0, 0

    strutRestLength = 0.6
    strutStiffness = 25.0
    strutDampingCompression = 5.1
    strutDampingRelaxation = 2.3
    frictionBreakout = 1000.0
    wheelRadius = 0.5
    rollInfluence = 0.1
    strutConnectionOffset = 0.0, 0.0, 1.4
}

collisionObject carWheelFrontLeft : carWheel
{
    steerable = true
}

collisionObject carWheelFrontRight : carWheel
{
    steerable = true
}

collisionObject carWheelBackLeft : carWheel
{
    steerable = false
}

collisionObject carWheelBackRight : carWheel
```

```
{
    steerable = false
}
```

Practically speaking the only collision shape that makes sense right now for the vehicle chassis is BOX because currently there is a known issue with MESH collision shapes.

In the initialize() method of your Game class, you can set a member variable for accessing the vehicle via PhysicsVehicle:

```
Node* carNode = _scene->findNode("carbody");
if (carNode && carNode->getCollisionObject()->getType() == PhysicsCollisionObject::VEHICLE)
{
    _carVehicle = static_cast<PhysicsVehicle*>(carNode->getCollisionObject());
}
```

Then, in the update() method of your Game class you need to call PhysicsVehcicle::update() with the various control inputs described below:

```
_carVehicle->update(elapsedTime, steering, braking, driving);
```

The steering parameter controls vehicle steering and has an expected range of -1 to +1. The braking parameter applies wheel brakes and has an expected range of 0 to 1. The driving parameter lumps together engine output and overall drivetrain, with an expected range of 0 to 1. Optional properties in the .physics definition for your vehicle give you greater control of the handling characteristics, and are described below in turn.

Overall Vehicle Controls

The following properties specify the vehicle's overall response to control inputs:

```
// Vehicle steering, braking, and powertrain
steeringGain     = <float>       // steering at full deflection
brakingForce     = <float>       // braking force at full braking
drivingForce     = <float>       // driving force at full throttle
```

This is an over-simplification of vehicle handling, and therefore IxGame provides further refinement as follows.

Steering Reduction at High Speed

Turning the steering wheel of a real car by 1 degree has a much different effect at 100 km/h than it does at 10 km/h. (Please do not attempt this). In a real vehicle, the "feel" of a steering wheel tends to stiffen as speed increases. In a racing game, we can approximate this effect by reducing the amount of authority at higher speeds. IxGame provides the following properties to control this effect:

```
// Steering gain reduction with speed (optional)
steerdownSpeed = <float>       // steering gain fades to this point
steerdownGain  = <float>       // gain value at that point (less than 1)
```

The gain at zero speed is always 1. The properties steerdownSpeed and steerdownGain specify a point of reduced gain, above which the gain remains constant. A steerdownGain of 1 effectively disables this feature.

Brake Reduction at High Speed

Due to imperfections in the simulated physics, full braking at high speeds can cause unexpected behavior. IxGame provides the following properties to reduce braking above a certain threshold speed:

```
// Brake force reduction at high speeds (optional)
brakedownStart = <float>        // braking fades above this speed
brakedownFull    = <float>        // braking is fully faded at this speed
```

Braking remains unaffected up to the speed specified by brakedownStart. Above that speed, braking fades and reaches zero at the speed specified by brakedownFull. An unreachably-large speed value for brakedownStart will effectively disable this feature.

Vehicle Acceleration, All-out Speed, and Aerodynamic Downforce

Driving force is currently simplified down to a single value that lumps together the engine and drivetrain. In the absence of a proper gearbox simulation IxGame provides the following properties for affecting all-out speed and bottom-end acceleration:

```
// Driving force reduction at high speeds (optional)
drivedownStart = <float>      // driving force fades above this speed
drivedownFull    = <float>      // driving force is fully faded at this speed

// Driving force boost at low speeds (optional)
boostSpeed          = <float>        // Boost fades to 1 at this point
boostGain           = <float>        // Boost at zero speed (greater than 1)
```

The first 2 properties allow you to reduce driving force at high speeds which limits the top speed of the vehicle. Above the speed specified by drivedownStart driving force begins to fade, and eventually reaches zero at the speed specified by drivedownFull, more or less. An unreachably-large value for drivedownStart will effectively disable this feature.

The last 2 properties allow you to increase acceleration at low speeds. boostGain specifies the gain at zero speed, so a value greater than 1 will increase vehicle acceleration from a standing start. This supplemental gain then fades to 1 at the speed specified by boostSpeed. A boostGain of 1 effectively disables this feature.

Racing cars typically make use of airfoils to produce a downward force at high speeds. This improves handling and performance. IxGame provides the following property to simulate this effect:

```
// Aerodynamic downforce effect (optional)
downforce           = <float>        // proportional control of downforce
```

The value of downforce controls the amount of downward force at a given speed. In particular, the value of this property represents the product of a reference area and an aerodynamic coefficient. However what's important is that this property acts as a constant of proportionality in computing the downward force as a function of speed. A value of 0 effectively disables this feature.

PhysicsVehicleWheel

Tire and suspension characteristics can be specified at each individual wheel as follows:

```
collisionObject <wheelID>
{
```

```
    type                        = VEHICLE_WHEEL

    steerable                   = <bool>          // indicates whether wheel is steerable
    wheelDirection              = <float, float, float> // direction of strut extension, in chassis space
    wheelAxle                   = <float, float, float> // direction of axle (spin axis), in chassis space
    strutConnectionOffset       = <float, float, float> // offset from default strut connection point
    strutRestLength             = <float>         // strut rest length
    strutTravelMax              = <float>         // maximum strut travel
    strutStiffness              = <float>         // strut stiffness, normalized to chassis mass
    strutDampingCompression     = <float>         // strut damping under compression, normalized to chassis mass
    strutDampingRelaxation      = <float>         // strut damping under relaxation, normalized to chassis mass
    strutForceMax               = <float>         // maximum strut force
    frictionBreakout            = <float>         // breakout friction
    wheelRadius                 = <float>         // wheel radius
    rollInfluence               = <float>         // how side friction affects chassis roll, normalized
}
```

IxGame automatically determines a default location on the chassis for the strut connection point based on the position of the wheel nodes relative to the car body. The strutConnectionOffset property allows you to specify an offset from the default. This is useful, for example, if the origin of the car body is not located at the center of the mesh:

```
strutConnectionOffset = 0.0, 0.0, 1.4
```