gameplay

Game Development Guide

Published: 2012-03-09
SWD-20120309110431151

Contents

Overview

1

The gameplay framework is an open-source, cross-platform gaming framework that makes it easy to learn to write 3-D mobile and desktop games using native C++. In this guide, we cover a top-down approach to teaching you the gameplay library, tools, and all the major parts of the APIs that are included in the framework. This guide covers a set of the C++ classes that you will be using to write your games.

Design goals and considerations

In creating the gameplay framework, the goal was not only to focus on creating a high performance native C++ game framework, but a clean, simple, and elegant architecture too. The framework uses a minimal set of cross-platform external dependencies and tries to take a more minimalist approach to designing the classes. This approach allows you to learn from the classes in the library and extend the framework to build your own game engine and tools from it. This framework is a good starting block for learning how to write cross-platform 3-D mobile and desktop games that gives you more time to focus on designing your game.

Why write another ‘game engine’?

We actually hope it will be considered more of a gaming framework, however in essence it really is still just the core components of a game engine. There are several reasons why the gameplay framework was developed.

First, most modern 3-D game engines, while sometimes free or often cheap, are closed source. Additionally they usually have licensing fees. The gameplay framework is free and open-source software under the Apache CXF license. We want more people to learn about the fact that cross-platform is a reality and building a good base lets you move forward to writing game titles.

Secondly, a game engine is not only about rendering. Yes, it’s a huge part but equally important are other aspects of game engines, such as audio, physics, animation, UI forms, particle emitters, scripting, and math. Searching on the web and trying to find information on how to fit these things into your games and game engine will probably lead you scattered all over the web with a lot of gaps in your learning. This framework will hopefully bring it all together for you.

Lastly, mobile is hot! The gameplay framework will have a lot of emphasis on gaming on mobile devices. Today it’s the biggest growing sector of the game industry. We still have platform support for major desktop platforms for both tooling and PC gaming. We think more focus should be on mobile gaming and learning to write games that can easily target the cross-platform mobile sector.

Building framework and platform support 2

To build the gameplay library, tools, and samples as well as write your own games, you will have to install and use the platform specific tools listed below. These are the supported environments for each specific operating system as a target platform.

Desktop OS	Tool	Development environment
Microsoft	Microsoft Visual Studio 2010	Windows XP/Windows 7 only
Apple	Apple Xcode 4	Mac OS X only

Mobile OS	Tool	Development environment
BlackBerry Tablet OS 2	BlackBerry Native SDK 2.0	Windows XP/Windows 7, Mac OS X, or Linux
Android OS 4	Android NDK 7	Windows XP/Windows 7, Mac OS X, or Linux
AppleiOS 5	AppleXcode 4	Mac OS X only

Project repository

Included in the project repository are the following notable folder and files:

Folder or File	Description
/external-deps	External dependency libraries
/gameplay	The gameplay library
/gameplay-api	Doxygen API Reference
/gameplay-docs	Documentation guides and tutorials
/gameplay-encoder	Game asset/content encoding tool
/gameplay-samples	Game samples
gameplay.xcworkspace	The Xcode workspace for developing for Mac OS X and iOS 5
gameplay.sln	The Microsoft Visual Studio solution for developing for Windows 7
gameplay-newproject.bat/.sh	The scripts to generate new cross platform game projects

Getting started on desktop

The quickest way to get started using the gameplay framework and tools is to simply start working in one of the desktop environments. We would recommend using either Microsoft Visual Studio 2010 Express or Professional on a Windows XP or Windows 7 operating systems, or Apple XCode 4 on a Apple Mac OS X operating system. Then just open either the Microsoft Visual Studio solution or XCode workspace and build and run the projects. These solutions/workspaces are set up by default to build all the projects needed and to run the samples you have selected as active.

Voila! You now have one of several simple interactive samples running on your desktop environment, which you can now explore and become more familiar with.

Game samples

The gameplay-docs folder contains additional tutorial documentation for our gameplay-samples. These are intended to go into more details for designing and coding games written using the framework. They all have a good starting point but have intentionally been left incomplete. Hopefully this will leave you with a feeling that will make you want to change the samples, complete them, and even make them more fun to interact and play with. They provide good starter code and a basis for you to explore various features in the framework. You can steal code snippets from the samples to help speed up the development cycle in your own games.

Creating new projects

For creating new cross-platform game projects, run the gameplay-newproject.bat/.sh script.

API reference

We firmly believe in making a very intuitive set of APIs that are as straight-forward and as consistent as possible. However, all of the classes in the framework have also been well documented with Doxygen formatting and will be constantly improved and updated iteratively throughout each release. This is to help you learn about what classes or sets of functions can be used and the recommended practices in using them.

We highly recommend reading the latest versions of the API Reference from the pre-generated HTML Doxygen API documentation in the gameplay-api folder. This will give you a deeper understanding of the C++ gameplay framework.

Getting started on mobile

Now that you are up and running on one of the Desktop/PC environments, we recommend you take this seriously and go mobile! Start off by downloading the Native Development Kits (NDK) for one of the various supported mobile operating system targets or set them all up.

In today’s mobile game market, cross-platform development is a reality. It is actually quite simple and easy using the gameplay framework to target a much wider device audience by just downloading them all.

Mobile setup instructions

Listed below are the basic setup instructions for downloading and installing the supported mobile platform and development environments for gameplay.

BlackBerry Native SDK for Tablet OS (BlackBerry PlayBook tablet)

1. Download and install the BlackBerry Native SDK 2.
2. Run the QNX Momentics IDE (Eclipse CDT based) and click **File** > **Import** > **Import Existing Projects**.
3. Import all the gameplay projects by selecting the repository project folder.
4. Set the active configuration to one of the Device-XXX or Simulator-XXX profiles.

5. Build and run any of the game samples.

Apple Xcode 4 (iPad tablets and iPhone devices)

1. Download and install Apple Xcode 4.
2. Open the gameplay.xcworkspace.
3. Change the active configuration to iOS Device, iPhone Simulator or iPad Simulator.
4. Build and run any of the game samples.

Google Android NDK 7 (Android tablets and devices)

1. Download and install Google Android NDK 7 and Android SDK.
2. Build the gameplay library using following steps:
   a Change to <gameplay-root>/gameplay/android folder.
   b Run the following command to build generate a build.xml file:

   > **android.bat update project -t 1 -p. -s**

   c Run the following command to build the gameplay library:

   > **ndk-build**

3. Build a sample game with following steps:
   a Change to the <gameplay-samples-root/sample-game-root>/android folder.
   b Run the following command to generate a build.xml file:

   > **android.bat update project -t 1 -p. -s**

   c Run the following command to build the gameplay sample:

   > **ndk-build**

4. Build a sample game with following steps:
   a Change to the <gameplay-samples-root/sample-game-root>/android folder.
   b Run the following command to generate a build.xml file:

   > **android.bat update project -t 1 -p. -s**

   c Run the following command to build the gameplay sample:

   > **ndk-build**

5. Connect the device and run the following command to deploy the game:

> **ant debug install**

Mobile platform considerations

Ensure you test early on the physical devices. Depending on the type of game you want to write and your design ideas, you'll want to get some idea of what type of performance you'll get with the game plan and prototypes you are working towards. Be careful and do not to rely on Desktop and Mobile Simulators as an indicator of performance or mobile device capabilities.

Game consoles and Linux distributions

We are very open to the idea that the gameplay framework can be extended to target system game consoles and various custom Linux environments. Currently there is no support for these systems.

However, please let us know if you have interest in supporting a popular game console or Linux distribution. If the quality of the port is good then we will host it for you!

Game assets and authoring 3

Game assets are extremely important for the quality of a good game. Not only do the game assets need to be fitted for the game design, but they also need to load as quick as possible and at the highest quality within the platform hardware limitations.

Binary game assets

A very practical way to ensure you're being efficient is to always package and load all your game assets as binary formats. Common assets include images, fonts, audio and 3-D scenes. Most game engines will always include some sort of authoring tool to allow developers to encode and process their content to be game-ready. The gameplay framework also includes an executable tool for this called the gameplay-encoder.

Using fonts and 3-D scenes

For fonts and 3-D scenes you will want to support industry standard file formats such as TrueType for fonts and popular modern 3-D scene formats such as COLLADA and FBX.

Although these formats are popular and have the widest support in tooling options, they are not considered efficient runtime formats. The gameplay library requires that you convert these formats to its documented gameplay binary format (.gpb) using the gameplay-encoder executable for loading them as binary files.

Building support for FBX

Although FBX is supported by the gameplay-encoder tools, FBX is not allowed to be re-distributed as part of our framework. However, it is free for you to use. Simply download the FBX SDK and then re-build the code in the gameplay-encoder using the USE_FBX preprocessor directive and ensure you include the header and library paths in the project to the FBX SDK paths.

Content pipeline

The content pipeline for fonts and scenes works like this:

1. Take any TrueType fonts or COLLADA/FBX scene files.
2. Run the gameplay-encoder executable passing in the font or scene file path and optional parameters to produce a gameplay binary version for the file. (.gpb)
3. Package your game and include the gameplay binary file as your binary game assets.
4. Load any binary game assets using the `gameplay::Package` class.

Using binary packages

Use the `gameplay::Package` class from your C++ game source code to easily load your encoded binary files as packages. The class offers methods to load both fonts and scenes. Scenes are loaded as a hierarchical structure of nodes, with various entities attached to them. These entities include things like mesh geometry or groups of meshes, cameras and lights. The `gameplay::Package` class also has methods to filter only the parts of a scene that you want to load.

Release mode assets

When releasing your game title, all of the images should be optimized and converted to the compressed texture format that is supported by OpenGL (ES). Audio should be encoded to save space on storage. For simplicity none of the gameplay-samples run using compressed textures.

Game

The `gameplay::Game` class is the base class for all your games created with the gameplay framework. You are required to extend this class using C++ and to override the initialize, finalize, update, and render methods that will be called as core game events for both lifecycle and frame events. 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 "gameplay.h"

using namespace gameplay;

class MyGame : public Game
{
    void initialize();
    void finalize();
    void update(long elapsedTime);
    void render(long elapsedTime);
};
```

The `Game::initialize()` and `Game::finalize()` methods are called when the game starts up and shuts down, respectively. They are the methods where 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 `gameplay::Platform` for each operating systems. 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.

Accessing the game instance

The `gameplay::Game` class can be accessed from anywhere in your game code. It implements a singleton design pattern meaning there is only one Game instance per application process. Call the static method `Game::getInstance()` to gain access to the instance of your game from any code.

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 (ES) 2.0 or Open AL 1.1. The Game's graphics devices 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 `gameplay::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 only be called 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.

Game services (controllers)

The `gameplay::Game` class also manages game services such as Audio/Animation and Physics Controllers and provides access to them directly using getter methods in the `gameplay::Game` class. 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 classes are being hosted by the `gameplay::Game` class and react on lifecycles event being handled in the game.

Game time and state

Once the instance of a `gameplay::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 for. 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 either be UNINITIALIZED, RUNNING or PAUSED.

Input and sensors

By creating your game and extending `gameplay::Game`, you'll be able to add all the required handlers of input events. Additionally there are methods on `gameplay::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 accelerometer state. The following illustrates overridden methods to handle input events:

```
#include "gameplay.h"

using namespace gameplay;

class MyGame : public Game
{
    ...

    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);
};
```

Handling input events

You have the opportunity, on either desktop platforms or mobile devices, to handle mouse events uniquely from the `Game::touchEvent()` method (this includes Bluetooth mouse support). However, this is not required and the default implementation of the `Game::mouseEvent()` method returns `false` where the user can allow mouse events to automatically be treated as touch events.

You can decide to optionally disable multi-touch support for games where you do not want this functionality. You can call `Game::setMultiTouch()` and pass in `false` to ensure the platform treats and handles touch events as single touches.

You can also call `Game::displayKeyboard()` to show or hide a virtual keyboard for platforms that support it. You'll want to integrate it into points in the game and user interfaces in the game where text input is required.

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

Sprites and fonts

Use the `gameplay::SpriteBatch` and `gameplay::Font` classes to integrate simple 2-D sprite and text rendering for both 2-D and 3-D games.

Binary encoding fonts

The first thing to do is to create or find a TrueType font that you want to use. Search the web and you can easily find sites to buy .TTF files or tools to make them.

Next, you'll want to binary encode your TrueType font to a binary format via gameplay-encoder to produce a binary file. To do this run the following command with your gameplay-encoder executable:

> **gameplay-encoder -f airstrip.ttf -s 28**

Drawing text and images

The following code sample illustrates how to render and image and text together:

```
void MyGame::initialize()
{
    // Create your sprite batch and font and associate resources
    _batch = SpriteBatch::create("res/image.png");
    _font = Font::create("res/airstrip28.gpb");
}

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

    // Draw your sprites (we will only draw one now
    _batch->begin();
    _batch->draw(Rectangle(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT),
                 Rectangle(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT), Vector4::one());
    _batch->end();

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

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

Batch, batch, batch

You'll notice that the `gameplay::SpritchBatch` and `gameplay::Font` code sequences above both have a common flow to them. The developer performs a call to `begin()` followed by drawing operations and finishing with a call to `end()`. This is to support batching or combining drawing operations into a single hardware rendering call.

Scene and nodes

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

The `gameplay::Scene` class is based on a hierarchical data structure that is often referred to as a scene graph. Using the `gameplay::Scene` and `gameplay::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 `gameplay::Camera` attached to it and the second node could have a `gameplay::Model` attached to it. The `gameplay::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 `gameplay::Node` class:

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

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 frame in 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 Maya

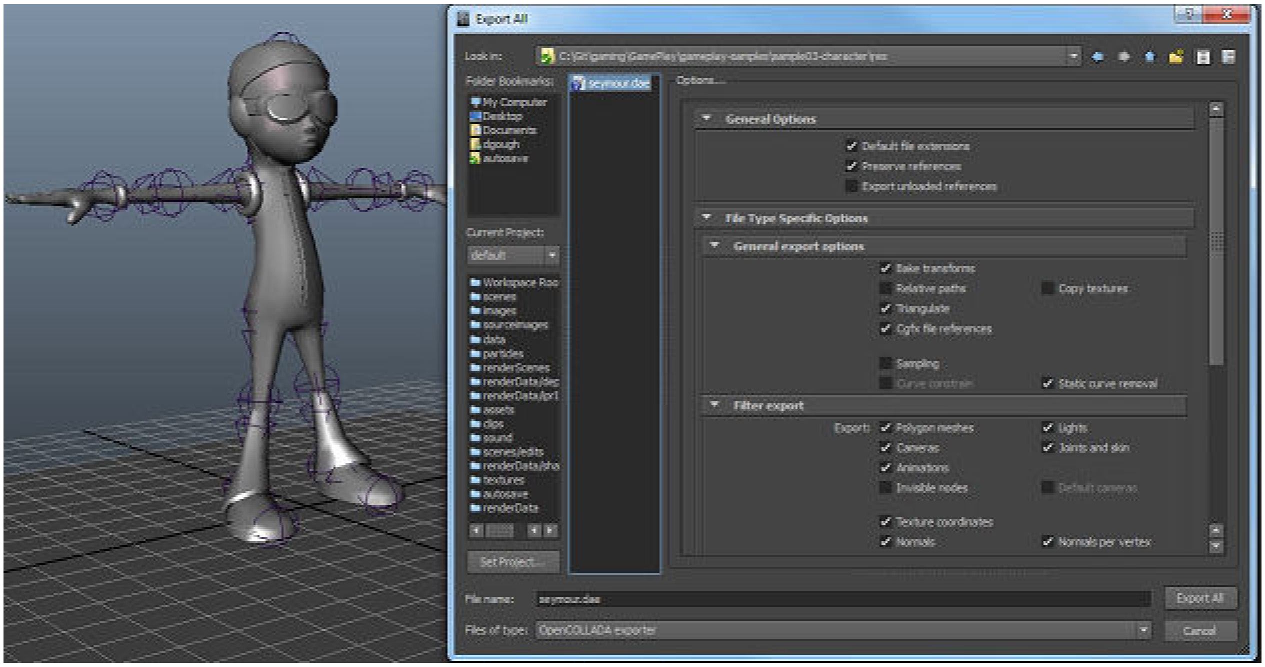
If you want to export your Maya scene to the COLLADA file format, it is suggested that you use the OpenCOLLADA plug-in. Once you install the plug-in, make sure it is enabled in Maya by going to Window -> Settings -> Plug-in Manager. Make sure that "COLLADAMaya.mll" is enabled.

When exporting a Maya scene to COLLADA:

- Choose file type: OpenCOLLADA exporter
- Make sure that Bake transforms is enabled.

When exporting a Maya scene to FBX:

- Choose file type: FBX exporter



Look in: C:\Git\gaming\GamePlay\gameplay-samples\sample03-character\res

seymour.dae

General Options
- ✓ Default file extensions
- ✓ Preserve references
- Export unloaded references

File Type Specific Options

General export options
- ✓ Bake transforms
- Relative paths
- Copy textures
- ✓ Triangulate
- ✓ Cgfx file references
- Sampling
- Curve constrain
- ✓ Static curve removal

Filter export

Export:
- ✓ Polygon meshes
- ✓ Lights
- ✓ Cameras
- ✓ Joints and skin
- ✓ Animations
- Invisible nodes
- Default cameras
- ✓ Texture coordinates
- ✓ Normals
- ✓ Normals per vertex

File name: seymour.dae

Files of type: OpenCOLLADA exporter

Exporting a 3-D scene from Blender

Blender supports exporting to the COLLADA and FBX file formats.

- File -> Export -> COLLADA (.dae)
- File -> Export -> Autodesk FBX (.fbx)

If you run into problems when using COLLADA files from Blender, try re-importing the COLLADA file back into Blender or Maya to see if there is a problem with the exported model.

Binary encoding a scene

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

Usage: gameplay-encoder [options] <filepath>

Example

Convert the COLLADA file "duck.dae" into gameplay binary file "duck.gpb".

> ***gameplay-encoder duck.dae***

Encoding an FBX file

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

Loading a scene

Using the `gameplay::Package` class, you can either load an entire scene or various parts of a scene into any existing scene. The `gameplay::Package` 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 gameplay binary file:

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

    // 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 ...
}
```

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 only to optimize performance. In this example we'll apply a rotation when the user has touched the screen or mouse button:

```
void MyGame::update(long elapsedTime)
{
    // Rotate the model
    if (!_touched)
        _modelNode->rotateY(MATH_DEG_TO_RAD(0.5f));
}
```

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

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

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 scenes hierarchical data structure is traversed and for each node in the scene the specified method is invoked as a callback.

```
void MyGame::render(long 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, &MeshGame::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;
}
```

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 such larger scenes would cause many models to be drawn unnecessarily. You can query a `gameplay::Node` class and retrieve a `gameplay::BoundingSphere` using `Node::getBoundingSphere()`. This bound represents a rough 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 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;
}
```

Model and mesh

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

Mesh geometry

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

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 `gameplay::Mesh` class maintains one vertex buffer to hold all the vertices and for each `gameplay::MeshPart`, an index buffer is used to draw the primitive shapes.

MeshSkin and joints

The `gameplay::Mesh` class supports an optional `gameplay::MeshSkin`. This is used when loading from models that represent characters in the game that have a Skeleton consisting of `gameplay::Joints` (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 gameplay 3-D framework supports a maximum of four blend weights per vertex. The `gameplay::MeshSkin` class holds and maintains a hierarchy of `gameplay:Joint` objects that can be transformed. A typical operation of this 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.

Lights

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

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

You can also programmatically create a light using the factory methods on the gameplay::Light class. Here is an example to add a directional light in your scene and bind the lighting information onto a models 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);
}
```

Pre-computed lighting maps

Adding lighting information into gameplay::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 lights additive color contributions using a process called 'baking'. This process allows the artist to direct the contributing light and color information into separate or combined texture so that this is not required during the rendering.

Directional lights

In most games you'll want to add a gameplay::Light class whose type is Light::DIRECTIONAL. This type of light is used as a 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 gameplay::Materials of objects that are dynamic or moving.

Point and spot lights

Due to the expensive processing overhead in using them, many games are designed to restrict point and spot light use to be static and to be initially baked into light and color maps. However, the point and spot light types add exceptional realism into games. Using them in either 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 gameplay .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 together. It should be noted that there is significant performance impact in doing this. For these cases, you'll usually want to restrict materials 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 `gameplay::Model` and then bind them to the `gameplay::Material` once they are found.

Materials and shaders 10

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

Using materials

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

RenderState and effects

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

Techniques

Since you can only bind one `gameplay::Material` per `gameplay::MeshPart`, an additional feature is supported to make it quick and easy to discretely change the way you render the parts at runtime. You can define multiple techniques by different names that can have completely different rendering techniques. Then by using `Material::setTechnique(const char* name)`, you can change the technique to be used at runtime. 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.

Creating materials

You can create a `gameplay::Material` from the simple `gameplay::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 `gameplay::Material` instance you'll want to get its parameters and then set or bind various values to them. To set a value, get the `gameplay::MaterialParameter` and then call the appropriate `setValue()` method on it. Setting material parameters values is most common in parameters that are based on values that are constants.

Example: 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 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).

Example: 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 they can usually be found in the `gameplay::RenderState` or `gameplay::Texture` class definitions:

```
material duck
{
    technique
    {
        pass 0
        {
            // shaders
            vertexShader = res/shaders/diffuse-specular.vsh
            fragmentShader = res/shaders/diffuse-specular.fsh

            // 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
            }
        }
    }
}
```

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 `gameplay::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.

Particle emitters

The `gameplay::ParticleEmitter` class defines all the information needed to simulate and render a system of particles. The emitter can be defined in various ways to represent smoke, steam, fire and other atmospheric effects such as rain and lightning. Once created, the emitter can be set on a `gameplay::Node` in order to follow an object or be placed within a scene.

Particles as sprites

A `gameplay::ParticleEmitter` always has a sprite/texture and a maximum number of particles that can be alive at any given time. After the emitter is created, these cannot be changed. Particles are rendered as camera-facing billboards using the emitter's sprite/texture. The emitter's sprite/texture properties determine whether the texture is treated as a single image, a texture atlas, or an animated sprite.

Particle properties

A `gameplay::ParticleEmitter` also has a number of properties that determine values assigned to the individual particles it emits. Scalar properties such as particle begin- and end-size are assigned within a minimum and maximum value; vector properties are assigned within the domain or space and are defined by a base vector and a variance vector.

The variance vector is multiplied by a random scalar between 1 and -1, and the base vector is added to this result. This allows an emitter to be created which emits particles with properties that are randomized, yet fit within a well-defined range. To make a property deterministic, simply set the minimum value to the same value as the maximum for that property or set its variance to a zero vector. To learn more about all different scalars, vector and rendering properties that can be set on a `gameplay::ParticleEmitter`, look at the C++ API.

Creating particle emitters

Use the `ParticlEmitter::create()` method to create an emitter from a particle file. The .particle file format and semantics are very similar to the .material file format. This is because it also leverages the `gameplay::Properties` file definition and supports all the properties supported in the C++ API for the `gameplay::ParticleEmitter` class.

Animated sprites for particles

It is very easy to make the particles animate through a list of images. Just make your images have a tile of sprite images and then modify the sprite’s base properties in the emitter to control the animation behavior.

You can then even do things like animate images of a 3-D dice using only 2-D images.

Physics

The gameplay 3-D framework supports a game service/controller that is the `gameplay::PhysicsController` class. The `gameplay::PhysicsController` maintains a physics 'world'. This world has gravity and will simulate the objects added to it.

The gameplay 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 `gameplay::PhysicsRigidBody` object representing the geometry or `gameplay::Model`. By attaching a rigid body to a `gameplay::Node`, the rigid body will be added to the physics world and the simulation will automatically update the node's transformation.

Creating a gameplay::PhysicsRigidBody

To create a rigid body, you first 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 like boxes and spheres you can programmatically create the rigid bodies by calling `Node::setRigidBody()` and passing in the desired shape type.

All other types of rigid bodies must be created using the `.scene` and `.rigidbody` 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
    {
        ...
        rigidbody = game.rigidbody#tree_mesh
    }
    ...
}


game.rigidbody:

rigidbody tree_mesh
{
    type = MESH
    mass = 15.0
    ...
}
```

Creating a PhysicsContraint

The gameplay 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 `gameplay::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
    }
    }
    ...
}
```

Handling collisions

The gameplay 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 `gameplay::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 gameplay::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)
{
    WARN_VARG("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);
}
```

Animation

Animation is a key component to bringing your game to life. Within gameplay there is support to create both property animations and character animations. The `gameplay::Game` class exposes a service called the `gameplay::AnimationController` which can be used to create animations on properties of classes that extend `gameplay::AnimationTarget`. Character animations from within the scene file are imported into the `AnimationController` when the `.gpb` file is loaded. All animations are stored on the controller and can be obtained by ID.

AnimationTargets

Both `gameplay::Transform` and `gameplay::MaterialParameter` are animation targets.

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

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

Creating property animations

Animations are created from the `gameplay::AnimationController`. The animation controller holds a reference to each `gameplay::Animation` it creates. Each reference can be queried by the animation's ID. The animation controller provides methods to create simple two key frame animations using `AnimationController::createAnimationFromTo()`, and `AnimationController::createAnimationFromBy()`. Multiple key frame sequences can be created from `AnimationController::createAnimation()`.

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

```
AnimationController* controller = Game::getAnimationController();
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* animation = controller->createAnimation("my_animation", enemyNode,
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 `gameplay::AnimationController`:

```
animation my_animation
{
    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* animation = controller->createAnimation("my_animation", enemyNode,
Transform::ANIMATE_TRANSLATE,    "sample.animation");
```

Curves

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

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.

These animations can be obtained by calling `AnimationController::getAnimation()` specifying the animation's ID.

AnimationClips

A `gameplay::AnimationClip` is created from the `gameplay::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", 3333L, 5167L);
elfRun->setRepeatCount(AnimationClip::REPEAT_INDEFINITE);
elfRun->setSpeed(1.5f);
```

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 the elf animation:

```
animation elf
{
    frameCount = 350
    clip elf_idle
    {
        begin =  0
        end = 75
        repeatCount = INDEFINITE
    }
    clip elf _walk
    {
        begin = 75
        end = 200
        repeatCount = INDEFINITE
    }
    clip elf_run
    {
        begin = 200
        end = 310
     repeatCount = INDEFINITE
        speed = 2.0
    }
    clip elf_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.

Animation blending

The `gamplay::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.

AnimationClip listeners

Animation events can be triggered on a `gameplay::AnimationClip` by registering instances of `gameplay::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 point 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.

Audio

You can easily integrate 3-D audio into your game using the audio services supported by gameplay. The framework uses a very traditional way of representing audio. The `gameplay::AudioController` manages all of the playing audio sources.

Creating an AudioSource

An `AudioSource` can be created from audio files or from a .audio property file. Ogg audio files are compressed so they use less memory than .wav files.

```
AudioSource* wheelsSound = AudioSource::create("res/longboard.wav");
AudioSource* backgroundMusic = AudioSource::create("res/music.ogg ");
```

Playing the AudioSource

The following example illustrates how to play audio:

```
wheelsSound->play();
```

Updating the AudioListener

By default, the `AudioListener` is bound to the active camera of the scene. You can manually bind the camera to the `gamplay::AudioListener` using `gameplay::AudioListener::setCamera()`.

Audio Properties

The `gameplay::AudioSource` class has methods for modifying the properties of the `AudioSource` such as pitch, gain, and velocity.

Audio sources can be loaded from (`.audio)` property files to make it easier to set these properties.

```
audio fireball
{
    path = res/audio/fireball.wav
    looped = false
    gain = 0.7
    pitch = 0.5
    velocity = 0.5 0.0 1.0
}
```

Binding an AudioSource to a node

An audio source can be bound to a `Node` in your scene using `Node::setAudioSource()`. The position of the audio source is automatically updated when the node is transformed.

Legal notice 15

Research In Motion Limited
295 Phillip Street
Waterloo, ON N2L 3W8
Canada

Research In Motion UK Limited
Centrum House
36 Station Road
Egham, Surrey TW20 9LF
United Kingdom

Published in Canada