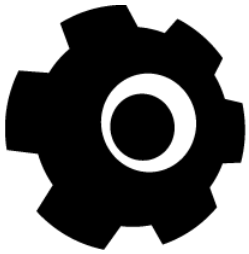


gameplay

Tutorial: Character

Contents

Controlling a character!

Object models are the building blocks of any game. They define how the game looks, and they're also fun to create. Sometimes though, you want to take your game to the next level. Adding animations and physics will allow you to put objects into your game that move realistically, adding depth and character to your game.

This tutorial will show you how to create, animate, and control a character in a game. It will look at the key steps of the modeling pipeline, from creating a model in a design tool, animating it, and importing it into an application using the gameplay library (https://github.com/blackberry/gameplay). Gameplay provides tools and APIs that make the process of importing and controlling your fully realized model easy and effective. It is recommended that you read the gameplay Development Guide before following this tutorial.

You will learn to

- Create game assets
- Export assets into commonly used formats
- Encode assets for use in your game
- Create the game!

This tutorial also explains how the relevant pieces of the gameplay library work. It is suggested that readers follow along with the sample03-character sample available on GitHub.

This tutorial is presented using the Autodesk Maya (http://usa.autodesk.com/maya/) tool, however many other tools can be utilized to create for your game.

Create game assets

A game isn't a game without cool visual elements that bring the player into the environment. One of the ways to keep a player interested in the game is to have a character that represents them, such as an avatar. To illustrate how you can create a character, we will take this funky guy here, named StevieGee, and bring him into a simple game:

The content creation process for bringing StevieGee into the game is as follows:

1. Create the game asset in a modeling tool
2. Export the asset out of the modeling tool
3. Encode the asset into a format that your game framework understands
4. Import the asset into your game

We will discuss content creation steps in-depth, but first, we'll describe how to create your game asset. You can use any number of tools to create your model, but here we'll use Autodesk Maya to illustrate the character creation process. The process for creating a game asset involves four steps:

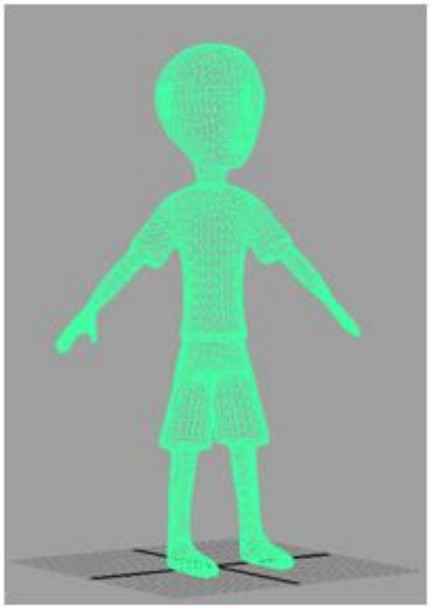
1. Creating the mesh
2. Binding the skeleton to the mesh
3. Animating the character
4. Lighting the character

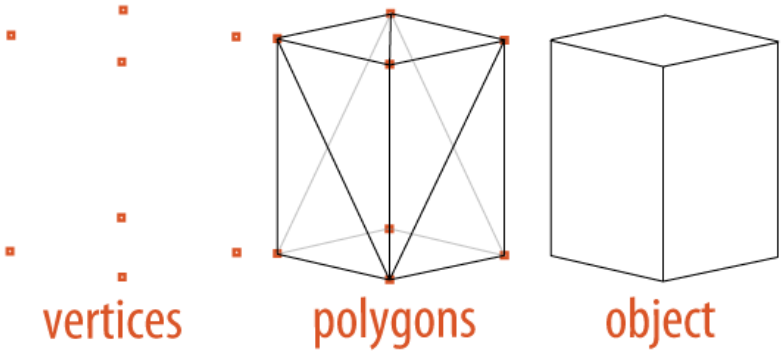
Creating the mesh

The first step in building a model is to create the 3-D surfaces that represent your object in the scene. To do this, you need some idea of what the objects will look like, and how they will move in order to accurately create them in the modeling tool. This requires some degree of imagination and artistic skill, as well as a familiarity with how modeling tools work. We'll give a very brief description of the concepts here but you are encouraged to browse the web and play around with tools to learn more details about object modeling.

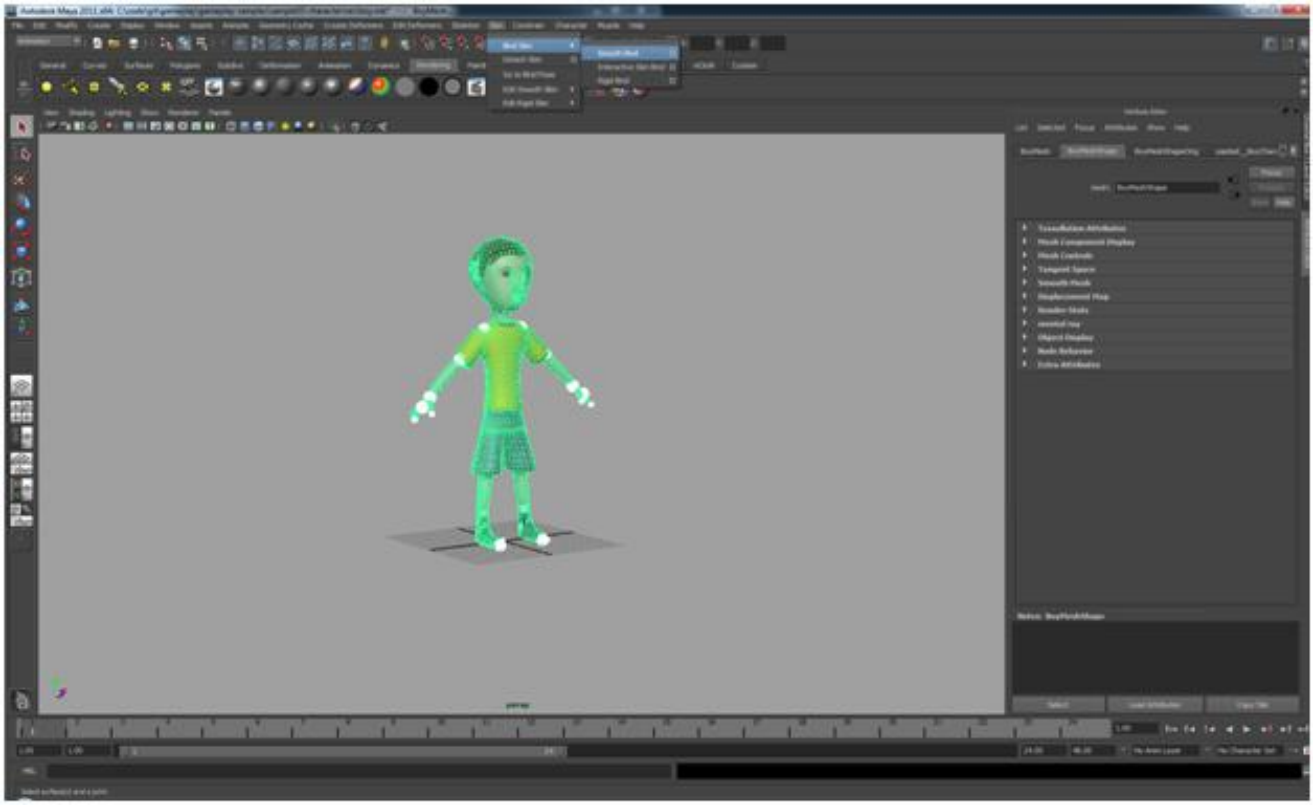
Typically, each surface on an object model is represented by a network of shapes that have three or more sides, these shapes are called *polygons*. Polygons are connected together to create an object model, or *mesh*, and the points where the polygons connect are called *vertices*.

The higher the amount of polygons, the more detailed you can make your objects. For this tutorial, we use the relatively simple StevieGee mesh.



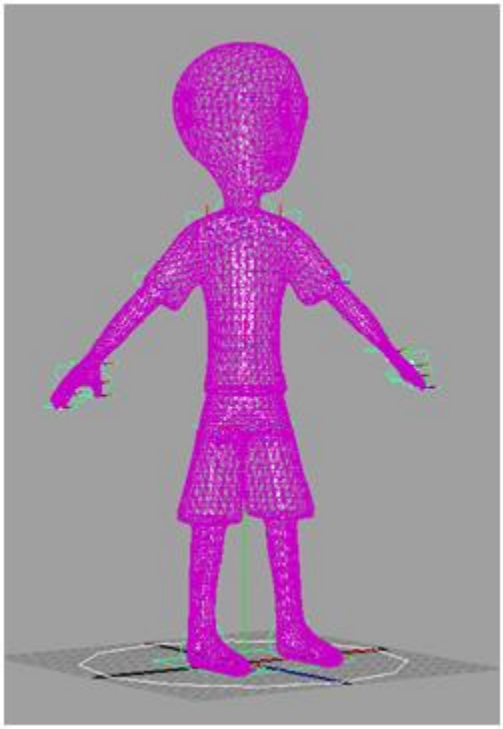


Of course, there are more ingredients involved in making a mesh look good, such as applying textures, materials, and lighting to the polygons, but we'll talk about those later in the tutorial. While we haven't gone into a lot of detail about creating meshes here (that could take several tutorials!), you can search the web to find your own content. There are plenty of freely available models out there, so it's just a matter of finding what you want. For now, we need to make the StevieGee mesh move, so that the player can interact with him in the game.



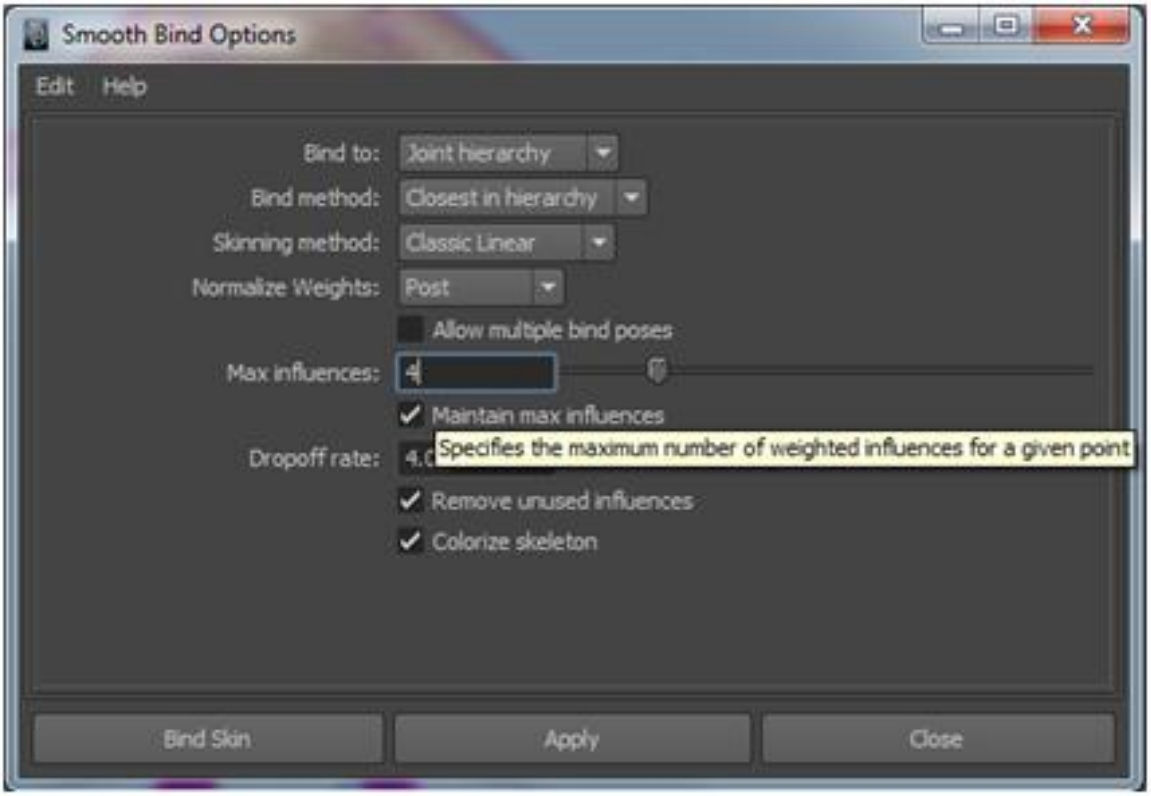
Binding the skeleton to the mesh

To make a mesh move, you need to give it a framework that defines which parts of the mesh are moveable and how those parts can be adjusted. A *skeleton* is used to define this framework. Just like a real human body, a mesh's skeleton has bones that define immovable sections and joints that define articulation points between bones. You set the position and rotation limits of joints on the skeleton so that they rotate in a convincing manner and, when they are animated in the game, you adjust these values to give the appearance of movement.



You combine the character's mesh with the character's skeleton through a process called *binding*. Binding a skeleton to a mesh defines a controlling mechanism that is used to deform the mesh according to the bone and joint properties of the skeleton. This means that when StevieGee's arm moves, you'll get the vertices and sides of the polygons around his elbow stretching or contracting, giving the appearance of realism. Usually the modeling tool calculates the deformation as part of the animation process, but you can further tweak how it looks. One such adjustment is to switch between the techniques used for binding, such as rigid and smooth binding. Here we'll use smooth binding because it allows for more than one joint in the skeleton to influence each vertex in the mesh, giving you a more realistic mesh deformation.

In Maya, the options used to smooth bind the skeleton to the mesh are located under the **Animation** view. Once in the **Animation** view, select both the model's mesh and skeleton. Then from the menu bar select **Skin** > **Bind Skin** > **Smooth Bind** and click the box next to **Smooth Bind** to bring up the **Smooth Bind Options**. To apply the bind, click the **Bind Skin** button in the **Smooth Bind Options** window.



Note: Our GPU shader programs only support a maximum of four weighted influences per joint.

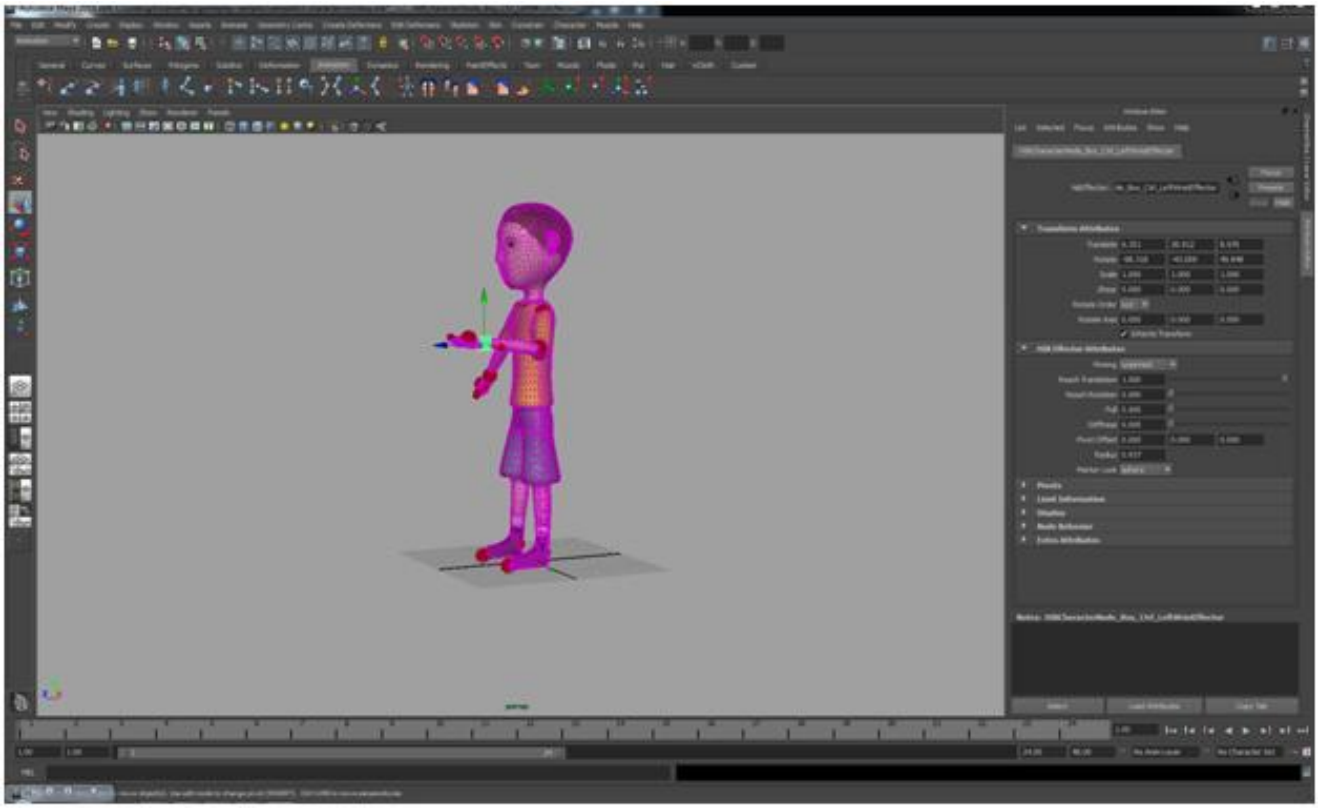
For more information on these properties, please visit the Smooth Bind section in the Maya User Guide.

Now, you have bound the skeleton to the mesh, but when you move the joints, the mesh might not deform in a realistic manner. This is where *influence objects* come into play. Influence objects are customized shapes you add to the skeleton to define deformation points beyond the joints. For example, you can position a spherical influence object near StevieGee's elbow so that the mesh bulges out as you move the joint. This can take a lot of fine-tuning and experimenting, but you'll see the results can add a lot to the overall impression of the model.

Animating the character

Now that StevieGee looks good, and has a backbone, it's time to get him moving. The act of animating a mesh is essentially defining a path that each joint in the mesh's skeleton can take for a particular movement. "Snapshots" are taken along that path to store as the skeleton's different states, and these can be played back to re-create the movement. Each snapshot is referred to as a *frame* and when a set of frames is played in sequence, you get an animation.

The rate at which frames are played back define how the animation looks and it is highly dependent on the medium where the animation is being played (television, film, video game, etc.). Typical frame rates are 24 or 30 frames per second (fps). The higher the frame rate, the more intensive the resource usage is. Because of this, it is typical for an animation to define *keyframes*. Keyframes are a subset of an animation's total amount of frames and specify the mesh's attributes at specific points in time. The application that does the rendering will generate the intermediate values between keyframes so the object moves smoothly and consistently. The more keyframes you have, the more you are directing the renderer along a specific path, so it takes a bit of experimentation to find the right balance between keyframes and performance. Keyframe animation is more traditional than some newer methods, like inverse kinematics, but it works for our purposes.



For StevieGee, we have defined five animation clips corresponding to his joints:

- Idle
- Walk
- Run
- Jump

It is essential to create your animations in place. This makes it easier to bring your animations into your game. To do this, ensure that you've selected the root or hip joints. Typically, you create the complete set of animations for your mesh sequentially along one timeline. The animations can then be indexed by frame and broken up into multiple animation *clips*. For example, we have set up StevieGee's animation so his idle pose is at frame 29 and his walk animation is from frame 42 to 66. Once you have defined these clips in your modeling tool, you can export them to an .animation file for use by your application. You can see these definitions in the boy.animation file located in the /res project directory. More information will be provided on the .animation file later in this document.

Lighting the character

You use light in a game to create variations in shading, giving added depth to the models in the scene. There are two parts to lighting a model inside a game - one is to define the location and properties of light sources in the environment, and the other is to define how the model itself reacts to light. Defining how the model reacts is accomplished by the properties of the *material* that is applied to the mesh. You can apply these properties either programmatically in the game, or by creating a materials file that can be loaded in at runtime. Either way, these properties are needed for the game to render the appearance of the model appropriately.

For StevieGee, we have decided to use material files that contain the information required to light the model, and the dance floor, correctly in the scene. The portion of the material file for the character looks like:

```
material boy : texturedTransparent
{
    technique
    {
        pass
        {
            defines = SKINNING;SKINNING_JOINT_COUNT 31;GLOBAL_ALPHA
            u_matrixPalette = MATRIX_PALETTE
            u_globalAlpha = 1.0

            sampler u_diffuseTexture
            {
                path = res/character.png
            }
        }
    }
}

material boyShadow : texturedTransparent
{
    technique
    {
        pass
        {
            sampler u_diffuseTexture
            {
                path = res/shadow.png
            }
        }
```

```
    }
}
```

For more information on the properties contained within a .material file, please consult the Materials and shaders section of the gameplay Development Guide.

Using the gameplay library, your game can load these material files dynamically at runtime, like this:

```
Material* meshMaterial = model->setMaterial("res /scene.material");
```

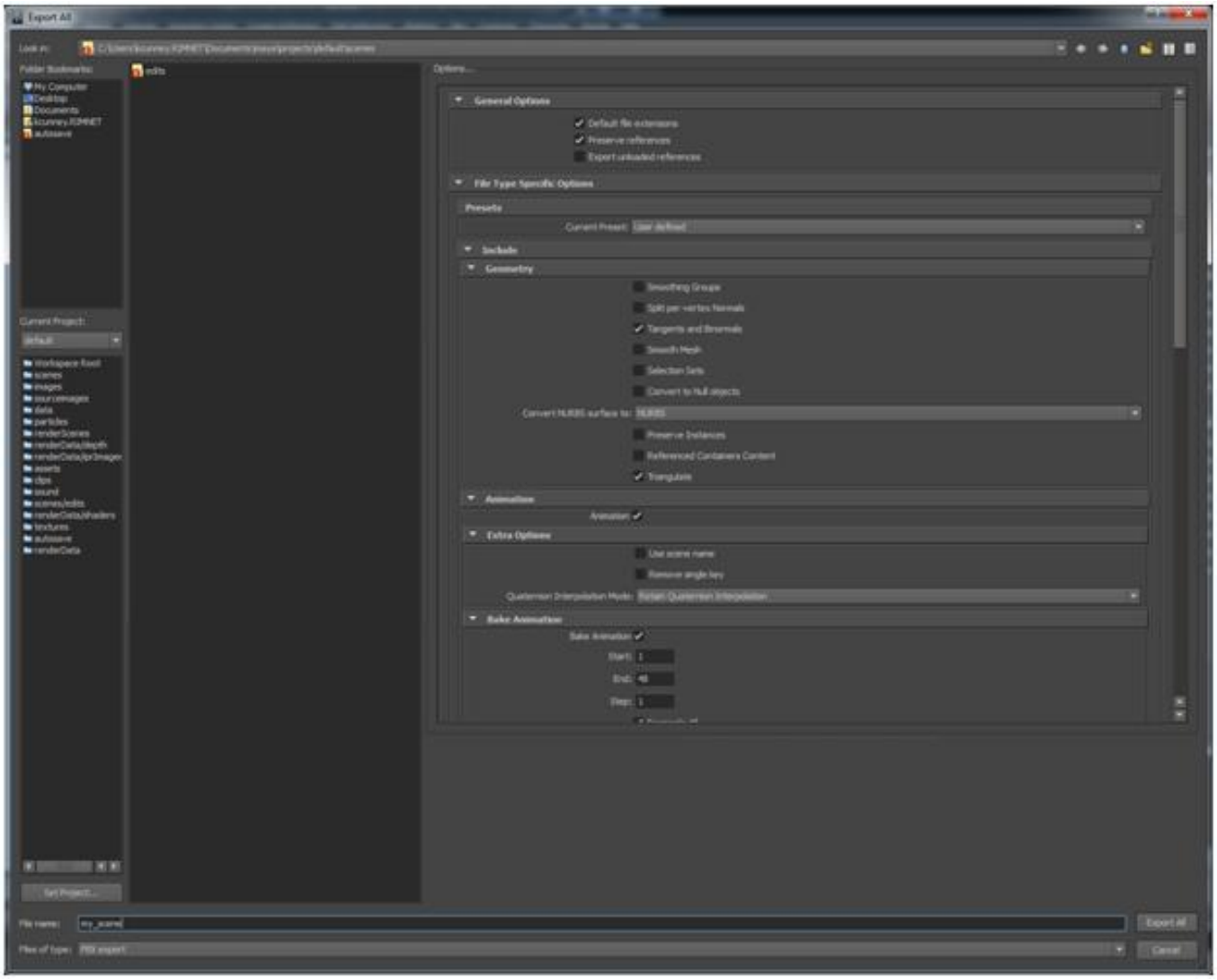
Export and encode your assets

Once you have completed your model, it needs to be taken out of the modeling tool so that you can bring it into your game. The method of exporting models varies from tool to tool, but it's always a good idea to export it into a commonly used format. This way, you have the most flexibility in terms of using the object.

Two commonly used exchange formats for digital assets are FBX and COLLADA. FBX is the interchange format between popular Autodesk tools like Maya and 3ds Max. COLLADA is a free, open source format that is popular within the interactive gaming industry. Using Maya as our tool, we'll show you how to export StevieGee into both formats.

Export to FBX

1) If you don't already have the FBX plug-in for Maya, you will need to download and install it in order to export to FBX. Please go to http://www.autodesk.com/fbx-downloads to download and install it.
2) To export your scene to .fbx, click **File > Export All**.
3) Change the Files of type: to FBX export.
4) Ensure the following additional options are checked off:
   a) Geometry > Triangulate
   b) Geometry > Tangents and Binormals
5) Name your file, and click **Export All**.



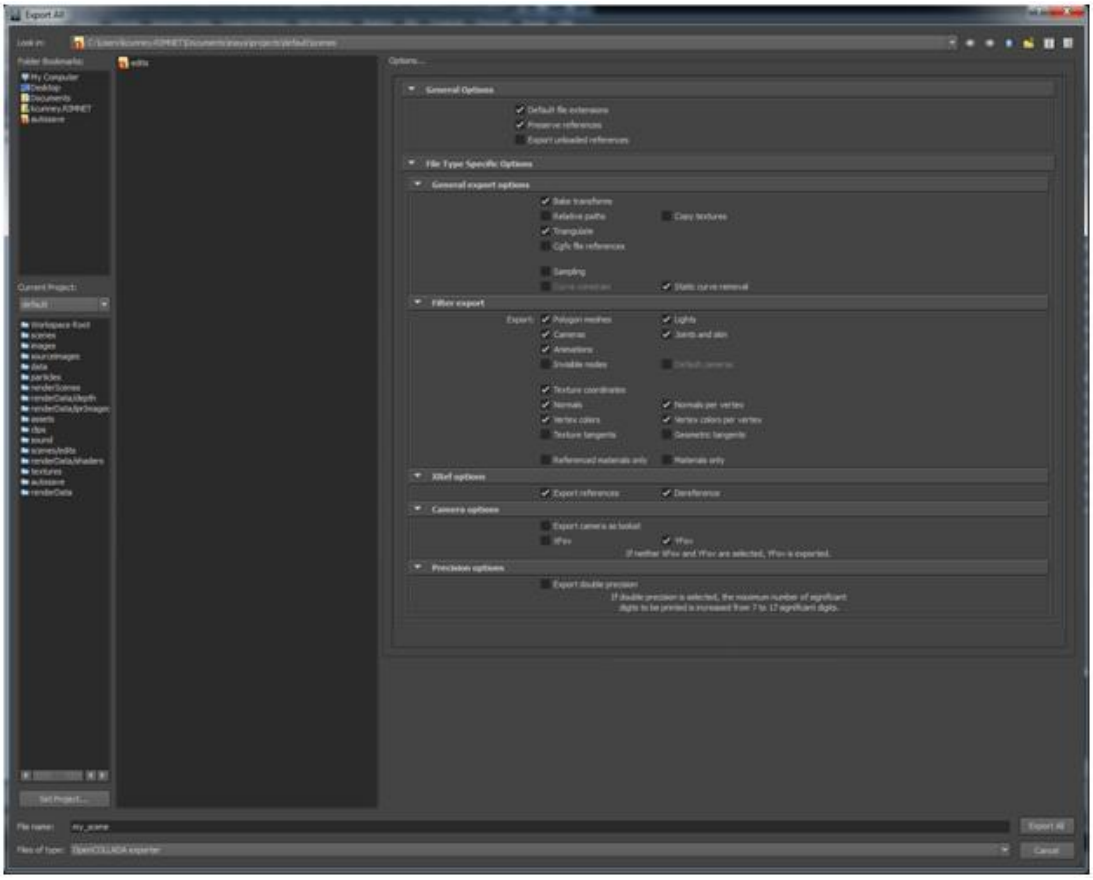
You now have your scene in FBX!

Export to COLLADA

The OpenCOLLADA plug-in for Maya currently only supports versions Maya 2011 and earlier.

1) If you don't have the OpenCOLLADA plug-in for Maya, please go to http://opencollada.com/download.html and download and install the latest version of the plugin. To enable the plugin in Maya:

a) On the Window menu, click Settings/Preferences > Plug-in Manager.
b) Ensure the checkboxes for **Loaded** and **Auto load** are selected beside



COLLADAMaya.mll

2) On the **File** menu, click **Export All**.
3) In the **Files of type** section, choose **OpenCOLLADA**.
4) In the options section, make sure the following options are selected:
   a) Bake transforms
   b) Triangulate
   c) Polygon meshes
   d) Joints and skin
   e) Animations
5) Name your scene and click **Export All**.

You now have your scene in OpenCOLLADA format.

Encode your asset to .GPB

Now that your model has been exported to a known format, there is one more step to complete in order to get it ready for use by your gameplay-enabled application. Your model needs to be converted into a format that the gameplay framework can use.

Adding Support for FBX to the gameplay-encoder

The gameplay encoder, by default, does not handle the encoding of FBX files. In order for it to encode FBX files, the encoder must be built with the Autodesk FBX SDK.

Here are the steps for readying the gameplay encoder to interpret FBX files:

1. First, you will need to download and install the Autodesk FBX SDK. This is available from http://www.autodesk.com/fbx-downloads.
2. Once the Autodesk FBX SDK is installed, open the gameplay-encoder project in Visual Studio and edit the project properties.
3. Add USE_FBX to the preprocessor definitions under Configuration Properties > C/C++ > Preprocessor.
4. Add the FBX SDK include directory to **Additional Include Directories** under **Configuration Properties > C/C++ > Genera**l (for example, `C:\Program Files\Autodesk\FBX\FbxSdk\2012.2\include`).
   Add the FBX library directory to the **Additional Library Directories** under **Configuration Properties > Linker > General** (for example, `C:\Program Files\Autodesk\FBX\FbxSdk\2012.2\lib\vs2010\x86`).
5. Add **fbxsdk-2012.2-mdd.lib** and **wininet.lib** to the **Additional Dependencies** under **Configuration Properties > Linker > Input** (for example,

fbxsdk-2012.2-mdd.lib;wininet.lib).

6. Add a post build event to copy the DLL under **Configuration Properties > Build Events > Post-Build Event** (for example, `copy /Y "C:\Program Files\Autodesk\FBX\FbxSdk\2012.2\lib\vs2010\x86\fbxsdk-2012.2d.dll" "$(TargetDir)"`).
7. Re-build the gameplay-encoder.

Encoding from COLLADA using OpenCOLLADA (.DAE)

To encode your COLLADA scene file into the gameplay binary format (.gpb) build the gameplay-encoder project, and run the following command:

```
gameplay-encoder –groupAnimations HIKCharacterNode_boy_Reference
movements "../../gameplay-samples/sample03-character/res/boy.dae"
```

Encoding from FBX (.FBX)

To encode your FBX scene file into the gameplay binary format (.gpb) build the gameplay-encoder project, and run the following command:

```
gameplay-encoder -groupAnimations HIKCharacterNode_boy_Reference
movements "../../gameplay-samples/sample03-character/res/boy.fbx"
```

As you can see, once FBX support is enabled in the gameplay encoder, encoding your FBX file is quite similar to encoding your DAE file. Let's take a look at some of the extra parameters passed into the encoder.

The `–groupAnimations` parameter says that you want to group all animations on HIKCharacterNode_boy_reference, and its children node into an animation called movements.

This encoding is necessary because the COLLADA, FBX, or, in the case of fonts, TrueType files are really just interchange formats, and are not usable by most game

libraries. Encoding them just follows the typical asset development pipeline.

.scene File

The `.scene` file is used to specify the characteristics of the scene. The `.scene` file must include the path to the .GPB file that the scene's content is loaded from. Additionally, the `.scene` file can include information about the active camera, global physical properties, constraints and animations, and various node attachments, such as particle emitters, audio sources, physics rigid bodies, and materials.

Here is the `.scene` file for our sample:

```
scene
{
    path = res/scene.gpb

    node boycharacter
    {
        character = res/scene.physics#boy
    }

    node boymesh
    {
        material = res/scene.material#boy
        dynamic = true
    }

    node boyshadow
    {
        material = res/scene.material#boyshadow
        transparent = true
        dynamic = true
    }

    node camera
    {
        ghostObject = res/scene.physics#camera
    }

    ...

    physics
    {
        gravity = 0.0, -9.8, 0.0
    }
}
```

(For the full sample please refer to res/scene.scene)

As you can see, we make use of node attachments like materials and rigid bodies. We also specify the global gravity constant to be applied within the scene. You can also define your physics constraints between rigid bodies here. The gameplay framework supports hinge, fixed, socket, spring, and generic six-degree-of-freedom type constraints.

Another thing to point out is the wildcard character (*). This is used as a convenience to apply the same settings on all nodes with IDs starting with the string provided in front of the wildcard. This saves you from having to define several nodes that will all have the same properties.

For more information on this, please consult the gameplay Development Guide.

.material File

The `.material` file is used to specify the techniques used to render the various materials within your scene. As you can see in the `.scene` file, the `.material` file is referenced in various nodes followed with a hash-tag. This simply states that the material for the node should utilize the technique with the specified namespace defined in the `.material` file. For more information on the .material file please consult the gameplay Development Guide.

For the reference sample please refer to res/scene.material.

.physics File

The `.physics` file is used to define the rigid body objects within the scene. Rigid bodies are used to represent the geometry of the objects within your scene in the

physics world. They serve as a definition of how each object interacts with other objects in the world. The physics system in gameplay supports rigid bodies of type box, sphere, capsule, mesh and terrain height field. All complex rigid bodies should be defined in the .physics file. For more information on the .physics file please consult the gameplay Development Guide.

For the reference sample please refer to res/scene.physics.

.animation File

The `.animation` file is used to define how you break up the animations of your scene into separate animation clips. We identify the animation by its ID and then namespace the clips we're going to define for the animation. At the very least for each clip definition you must include the begin and the end frame. Here we have also included the `repeatCount` property to specify how many times the animation clip should repeat. We have used the `INDEFINITE` flag for cycle animations like walk, and run, and a repeat once for one-off animations like jump. Additionally, you could define properties on the clip, such as `speed`.

```
animation boyAnimation
{
    frameCount = 1100
       clip idle
    {
        begin = 27
        end = 167
        repeatCount = INDEFINITE
    }
    clip walking
    {
        begin = 274
        end = 298
        repeatCount = INDEFINITE
    }
    clip running
    {
        begin = 331
        end = 346
        repeatCount = INDEFINITE
    }
```

```
    clip jump
    {
        begin = 473
        end = 486
        repeatCount = 1
        speed = 0.4
    }
}
```

For the full sample please refer to res/boy.animation

Finally, you can bring you can bring your animation clips into your game as follows:

```
_animation->createClips("res/boy.animation");
```

For more information on the .animation file please consult the gameplay Development Guide.

Loading game content

Before we get into the actual code of the game, let's look at what we're trying to accomplish:

- Bring StevieGee into a scene
- Control StevieGee's movements using touch inputs

Pretty simple, right? To create the game, we'll follow the standard lifecycle used for any game: initialize, update, render and finalize. The gameplay library provides the underlying framework for your application, so all you need to worry about at runtime is overriding the appropriate functions and callbacks to handle inputs. These methods are called automatically by the gameplay library at the application's frame rate. This allows you, as the game developer, to cleanly separate code for both handling updates to your game state and to render/draw your games visuals using a variety of built-in graphics classes.

The first thing you should do is subclass `Game` in `CharacterGame.h` :

```
class CharacterGame: public Game, public AnimationClip::Listener
{
    ...

   /**
     * @see Game::initialize
     */
    void initialize();

    /**
     * @see Game::finalize
     */
    void finalize();

    /**
     * @see Game::update
     */
    void update(long elapsedTime);
```

```
    /**
     * @see Game::render
     */
    void render(long elapsedTime);

    ...
}
```

Initializing the game

For the main program, we need to bring the visual elements we've created into the game. This is done in `CharacterGame::initialize()` function.

```
void CharacterGame::initialize()
{
    ...

    // Load scene.
    _scene = Scene::load("res/scene.scene");

    // Update the aspect ratio for our scene's camera
    // to match the current device resolution.
    _scene->getActiveCamera()->setAspectRatio((float)getWidth() /
                                              (float)getHeight());

    // Initialize the physics character.
    initializeCharacter();

    // Initialize the gamepad.
    ...
}
```

In this method, the first thing we do is load the .scene into the application using the `Scene::load(const char*)` function. As stated earlier, the scene file contains the references to the gameplay binary to load, as well as the active camera, animations, material definitions, physics properties and various node attachments.

```
_scene = Scene::load("res/scene.scene");
```

In order for the player to have a viewport into the scene, we need to define a camera. Here, we take the existing camera defined in the model file. We have

defined a perspective camera that makes objects in the distance appear smaller than objects closer by. This camera is defined to have an aspect ratio set by the game window's width and height.

```
_scene->getActiveCamera()->setAspectRatio((float)getWidth() / (float)getHeight());
```

Next, we call `CharacterGame::initializeCharacter ()`, where we obtain the `PhysicsCharacter` object for StevieGee along with the nodes for his mesh and his shadow. Then we store alpha parameter from StevieGee's material to allow us to draw him partially transparent when the camera is close up. Following that, we get the clips defined in `boy.animation` and load them into the `_animation` class variable. The last thing this method does is set the default animation to that defined by idle and sets it to play in an indefinite loop.

```
void CharacterGame::initializeCharacter()
{
    Node* node = _scene->findNode("boycharacter");

    // Store the physics character object.
    _character = static_cast<PhysicsCharacter*>(node->getCollisionObject());

    // Store character nodes.
    _characterMeshNode = node->findNode("boymesh");
    _characterShadowNode = _scene->findNode("boyshadow");

    // Store the alpha material parameter from the character's model.
    _materialParameterAlpha = _characterMeshNode->getModel()->getMaterial()->
                                            getTechnique((unsigned int)0)->
                                            getPass((unsigned int)0)->
                                            getParameter("u_globalAlpha");

    // Load character animations.
    _animation = node->getAnimation("movements");
    _animation->createClips("res/boy.animation");
    _jumpClip = _animation->getClip("jump");
    _jumpClip->addListener(this, _jumpClip->getDuration() - 250);

    // Start playing the idle animation when we load.
    play("idle", true);
}
```

The `PhysicsCharacter` class is a convenience class that is used to control the

movement and collisions of a character model within a game. It interacts with the physics system to handle gravity or any collisions, as well as controlling the character's movements. For additional information on `PhysicsCharacter` please consult the gameplay Development Guide.

Using the `scene.scene` file (see previous snippet) and the `scene.physics` file, we also define a collision object for the camera node. This will help position the camera so that it does not become occluded by a wall or some other object in the scene.

```
...

ghostObject camera
{
    type = SPHERE
    radius = 0.5
}

...
```

The last thing we do is create a Gamepad object and position its joystick and buttons on the screen. This Gamepad implementation will be used to control our character's walk, run and jump animations. For more information on the Gamepad, please consult the gameplay Development Guide.

Updating the game state

The `update()` method is utilized to apply your game logic, such as controlling your character. Here is the implementation of `update()` method for the game:

```
void CharacterGame::update(long elapsedTime)
{
    Vector2 direction;

    if (_gamepad->getButtonState(0) == Gamepad::BUTTON_PRESSED)
    {
        // Jump while the gamepad button is being pressed
        jump();
    }
    else if (_gamepad->isJoystickActive(0))
```

```
{
    // Get joystick direction
    direction = _gamepad->getJoystickState(0);
}
else
{
    // Construct direction vector from keyboard input
    if (_keyFlags & NORTH)
        direction.y = 1;
    else if (_keyFlags & SOUTH)
        direction.y = -1;
    if (_keyFlags & EAST)
        direction.x = 1;
    else if (_keyFlags & WEST)
        direction.x = -1;

    direction.normalize();
    if ((_keyFlags & RUNNING) == 0)
        direction *= 0.5f;
}

// Update character animation and velocity
if (direction.isZero())
{
    play("idle", true);
    _character->setVelocity(Vector3::zero());
}
else
{
    bool running = (direction.lengthSquared() > 0.75f);
    float speed = running ? RUN_SPEED : WALK_SPEED;

    play(running ? "running" : "walking", true, 1.0f);

    // Orient the character relative to the camera
    // so he faces the direction we want to move.
    const Matrix& cameraMatrix = _scene->getActiveCamera()->getNode()->
                                                            getWorldMatrix();
    Vector3 cameraRight, cameraForward;
    cameraMatrix.getRightVector(&cameraRight);
    cameraMatrix.getForwardVector(&cameraForward);

    // Get the current forward vector for the mesh
    // node (negate it since the character was modelled facing +z)
    Vector3 currentHeading(-_characterMeshNode->getForwardVectorWorld());

    // Construct a new forward vector for the mesh node
    Vector3 newHeading(cameraForward * direction.y + cameraRight * direction.x);

    // Compute the rotation amount based on the difference
    // between the current and new vectors
    float angle = atan2f(newHeading.x, newHeading.z) -
                  atan2f(currentHeading.x, currentHeading.z);
    if (angle > MATH_PI)
        angle -= MATH_PIX2;
```

```
        else if (angle < -MATH_PI)
            angle += MATH_PIX2;
        angle *= (float)elapsedTime * 0.001f * MATH_PIX2;
        _characterMeshNode->rotate(Vector3::unitY(), angle);

        // Update the character's velocity
        Vector3 velocity = -_characterMeshNode->getForwardVectorWorld();
        velocity.normalize();
        velocity *= speed;
        _character->setVelocity(velocity);
    }

    // Adjust camera to avoid it from being obstructed
    // by walls and objects in the scene.
    adjustCamera(elapsedTime);

    // Project the character's shadow node onto the surface directly below him.
    PhysicsController::HitResult hitResult;
    Vector3 v = _character->getNode()->getTranslationWorld();
    if (getPhysicsController()->rayTest(Ray(Vector3(v.x, v.y + 1.0f, v.z),
                                            Vector3(0, -1, 0)), 100.0f, &hitResult))
    {
        _characterShadowNode->setTranslation(Vector3(hitResult.point.x,
                                                     hitResult.point.y + 0.1f,
                                                     hitResult.point.z));
    }
}
```

In this method, we determine what animation our character StevieGee should currently be playing, as well as his directional movement. We do this by polling the state of the gamepad's controls. Here we've defined the joystick to control the character's movement around the scene, and the gamepad's button to make the character jump.

The last thing we do is call `adjustCamera(elapsedTime)`. This method will position our camera so it is not occluded by another object within the scene.

Rendering the Scene

The `render()` method is simply used to draw your scene.

```
void CharacterGame::render(long elapsedTime)
{
    // Clear the color and depth buffers.
```

```
    clear(CLEAR_COLOR_DEPTH, Vector4(0.41f, 0.48f, 0.54f, 1.0f), 1.0f, 0);

    // Draw our scene, with separate passes for opaque and transparent objects.
    _scene->visit(this, &CharacterGame::drawScene, false);
    _scene->visit(this, &CharacterGame::drawScene, true);

    // Draw debug info (physics bodies, bounds, etc).
    ...

    // Draw gamepad for touch devices.
    _gamepad->draw(Vector4(1.0f, 1.0f, 1.0f, 0.7f));

    // Draw FPS
    ...
}
```

The first thing we do is clear the color and depth buffers so we can start with a clean buffer to draw into, then we draw the scene by calling `visit()` on the scene object. This will traverse the scenes hierarchy, and for each node in the scene callback on the user specified method. Here we get it to call back on the `drawScene()` function, which checks to see if the node it is passed has a model, and if it has, draws it.

```
bool CharacterGame::drawScene(Node* node, bool transparent)
```

Touch

This game is touch input only, so we will override the `Game::touchEvent()` method to process the different types of touch inputs.

```
void CharacterGame::touchEvent(Touch::TouchEvent evt, int x, int y,
                               unsigned int contactIndex);
```

The `touchEvent()` function is able to handle three types of `TouchEvent`:

- `TOUCH_PRESS` - This represents the initial contact of a finger to the screen.
- `TOUCH_RELEASE` - This represents the end of a contact of a finger to the screen.
- `TOUCH_MOVE` - This handles any gestures on the screen. Here we rotate

StevieGee by figuring out where on the screen the move event occurred (using the x input parameter), calculating the delta between that and StevieGee's current rotation, and calling the `_modelNode->rotateY()` method to perform the rotation by the new delta

Conveniently, we have chosen to use the `Gamepad` implementation to handle the touch inputs for our game. So in here we simply call `_gamepad->touchEvent()` with the provided parameters. The joystick will allow the user to move StevieGee around the scene. Buttons will make him jump and we rotate him about his Y-axis so you can appreciate his moves from all angles.

Try it yourself

That's it, that's the full game that you can now build, deploy, and try out for yourself!

After you've run the game a few times, try changing the code and see what you can come up with. Some suggestions are:

- Create different animations for StevieGee, like making him wave or point.
- Add audio to the game for some atmospheric effects.

Once you're comfortable with this, try finding new meshes on the web (or even create your own from scratch) and load them into the game. See what gameplay can do!

License

The project is open sourced under the Apache 2.0 license.

Disclaimer

This documentation including all documentation incorporated by reference herein such as documentation provided or made available at www.blackberry.com/go/docs is provided or made accessible "AS IS" and "AS AVAILABLE" and without condition, endorsement, guarantee, representation, or warranty of any kind by Research In Motion Limited and its affiliated companies ("RIM") and RIM assumes no responsibility for any typographical, technical, or other inaccuracies, errors, or omissions in this documentation. In order to protect RIM proprietary and confidential information and/or trade secrets, this documentation may describe some aspects of RIM technology in generalized terms. RIM reserves the right to periodically change information that is contained in this documentation; however, RIM makes no commitment to provide any such changes, updates, enhancements, or other additions to this documentation to you in a timely manner or at all.

This documentation might contain references to third-party sources of information, hardware or software, products or services including components and content such as content protected by copyright and/or third-party web sites (collectively the "Third Party Products and Services"). RIM does not control, and is not responsible for, any Third Party Products and Services including, without limitation the content, accuracy, copyright compliance, compatibility, performance, trustworthiness, legality, decency, links, or any other aspect of Third Party Products and Services. The inclusion of a reference to Third Party Products and Services in this documentation does not imply endorsement by RIM of the Third Party Products and Services or the third party in any way.

EXCEPT TO THE EXTENT SPECIFICALLY PROHIBITED BY APPLICABLE LAW IN YOUR JURISDICTION, ALL CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS, OR WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY CONDITIONS, ENDORSEMENTS, GUARANTEES, REPRESENTATIONS OR WARRANTIES OF DURABILITY, FITNESS FOR A PARTICULAR PURPOSE OR USE, MERCHANTABILITY, MERCHANTABLE QUALITY,

NON-INFRINGEMENT, SATISFACTORY QUALITY, OR TITLE, OR ARISING FROM A STATUTE OR CUSTOM OR A COURSE OF DEALING OR USAGE OF TRADE, OR RELATED TO THE DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN, ARE HEREBY EXCLUDED. YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY STATE OR PROVINCE. SOME JURISDICTIONS MAY NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES AND CONDITIONS. TO THE EXTENT PERMITTED BY LAW, ANY IMPLIED WARRANTIES OR CONDITIONS RELATING TO THE DOCUMENTATION TO THE EXTENT THEY CANNOT BE EXCLUDED AS SET OUT ABOVE, BUT CAN BE LIMITED, ARE HEREBY LIMITED TO NINETY (90) DAYS FROM THE DATE YOU FIRST ACQUIRED THE DOCUMENTATION OR THE ITEM THAT IS THE SUBJECT OF THE CLAIM.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, IN NO EVENT SHALL RIM BE LIABLE FOR ANY TYPE OF DAMAGES RELATED TO THIS DOCUMENTATION OR ITS USE, OR PERFORMANCE OR NON-PERFORMANCE OF ANY SOFTWARE, HARDWARE, SERVICE, OR ANY THIRD PARTY PRODUCTS AND SERVICES REFERENCED HEREIN INCLUDING WITHOUT LIMITATION ANY OF THE FOLLOWING DAMAGES: DIRECT, CONSEQUENTIAL, EXEMPLARY, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR AGGRAVATED DAMAGES, DAMAGES FOR LOSS OF PROFITS OR REVENUES, FAILURE TO REALIZE ANY EXPECTED SAVINGS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF BUSINESS OPPORTUNITY, OR CORRUPTION OR LOSS OF DATA, FAILURES TO TRANSMIT OR RECEIVE ANY DATA, PROBLEMS ASSOCIATED WITH ANY APPLICATIONS USED IN CONJUNCTION WITH RIM PRODUCTS OR SERVICES, DOWNTIME COSTS, LOSS OF THE USE OF RIM PRODUCTS OR SERVICES OR ANY PORTION THEREOF OR OF ANY AIRTIME SERVICES, COST OF SUBSTITUTE GOODS, COSTS OF COVER, FACILITIES OR SERVICES, COST OF CAPITAL, OR OTHER SIMILAR PECUNIARY LOSSES, WHETHER OR NOT SUCH DAMAGES WERE FORESEEN OR UNFORESEEN, AND EVEN IF RIM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN YOUR JURISDICTION, RIM SHALL HAVE NO OTHER OBLIGATION, DUTY, OR LIABILITY WHATSOEVER IN CONTRACT, TORT, OR OTHERWISE TO YOU INCLUDING ANY LIABILITY FOR NEGLIGENCE OR STRICT LIABILITY.

THE LIMITATIONS, EXCLUSIONS, AND DISCLAIMERS HEREIN SHALL APPLY: (A) IRRESPECTIVE OF THE NATURE OF THE CAUSE OF ACTION, DEMAND, OR ACTION BY YOU INCLUDING BUT NOT LIMITED TO BREACH OF CONTRACT, NEGLIGENCE, TORT, STRICT LIABILITY OR ANY OTHER LEGAL THEORY AND SHALL SURVIVE A FUNDAMENTAL BREACH OR BREACHES OR THE FAILURE OF THE ESSENTIAL PURPOSE OF THIS AGREEMENT OR OF ANY REMEDY CONTAINED HEREIN; AND (B) TO RIM AND ITS AFFILIATED COMPANIES, THEIR SUCCESSORS, ASSIGNS, AGENTS, SUPPLIERS (INCLUDING AIRTIME SERVICE PROVIDERS), AUTHORIZED RIM DISTRIBUTORS (ALSO INCLUDING AIRTIME SERVICE PROVIDERS) AND THEIR RESPECTIVE DIRECTORS, EMPLOYEES, AND INDEPENDENT CONTRACTORS.

IN ADDITION TO THE LIMITATIONS AND EXCLUSIONS SET OUT ABOVE, IN NO EVENT SHALL ANY DIRECTOR, EMPLOYEE, AGENT, DISTRIBUTOR, SUPPLIER, INDEPENDENT CONTRACTOR OF RIM OR ANY AFFILIATES OF RIM HAVE ANY LIABILITY ARISING FROM OR RELATED TO THE DOCUMENTATION.

Prior to subscribing for, installing, or using any Third Party Products and Services, it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers might not offer Internet browsing functionality with a subscription to the BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with RIM's products and services may require one or more patent, trademark, copyright, or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use Third Party Products and Services and if

any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with RIM's products and services are provided as a convenience to you and are provided "AS IS" with no express or implied conditions, endorsements, guarantees, representations, or warranties of any kind by RIM and RIM assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with RIM.

Certain features outlined in this documentation require a minimum version of BlackBerry® Enterprise Server, BlackBerry® Desktop Software, and/or BlackBerry® Device Software.

The terms of use of any RIM product or service are set out in a separate license or other agreement with RIM applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY RIM FOR PORTIONS OF ANY RIM PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.

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

Published in Canada