Ride the asphalt wave!

GamePlay

Tutorial

BlackBerry

Published: 2011-11-03
SWD-1891766-1103121557-001

Contents

Ride the asphalt wave!

1

In this tutorial, we're going to have a close look at a little sample application that simulates a longboard moving along the pavement. The application should take input from the player and provide some sort of visual and audible feedback. How would you design and build this game? Here, we'll discuss one approach.

The Longboard sample uses the GamePlay library, which is a set of C++ classes that are designed to help you get started creating native games for the tablet. Although the classes in the GamePlay library are designed to make it particularly easy to create games for the tablet, they are also designed to make it easy to port your games to other platforms.

You will learn to

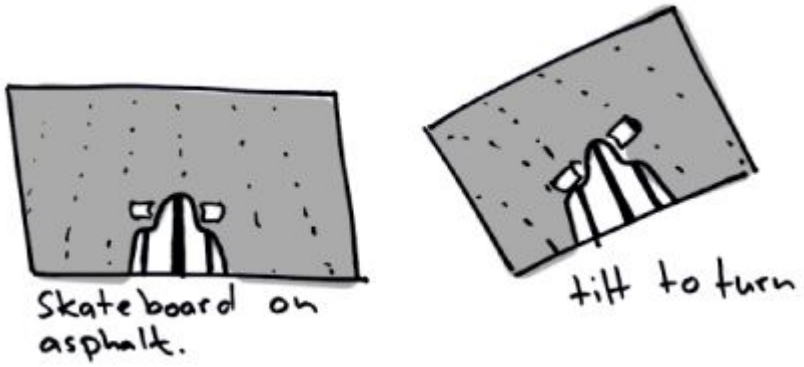
- Design the game
- Set up and initialize the game
- Update and render the game

It is suggested that readers follow along with the code in the Longboard sample.

Designing the game

2

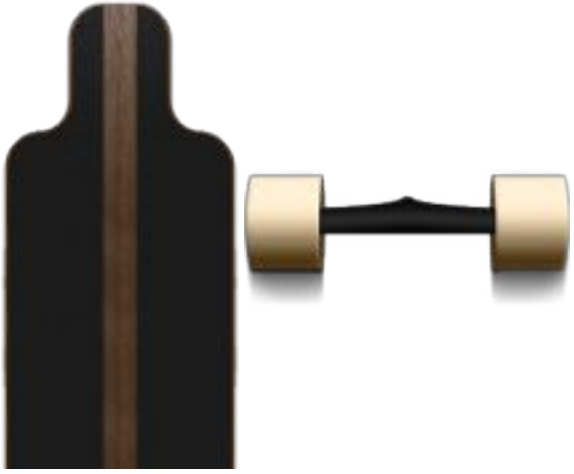
For this game, the longboard doesn't appear to move until the player provides input to control the longboard's speed or tilt. Tilting the device along the x-axis controls the speed of the board and tilting it along the z-axis (like a steering wheel) controls the direction of the board. If we wanted to sketch this out on paper, it would look like:



To keep the game simple, it will not track time, score, or apply any sort of limitation on the player. The longboard will simply move according to the player's inputs and visual and audible feedback will be provided.

The longboard

The longboard itself is broken up into two components, the deck and the wheels. We will make a decision here to create graphics files that represent these elements and apply them to game objects (meshes) programmatically. The position and shape of these objects will be adjusted during game time to give the appearance of motion and speed without actually moving them in 3-D space.



We will also apply a gradient to the overall scene to create an interesting lighting effect that focuses attention on the longboard.

The ground

Again, to keep things simple, the ground that the longboard rides upon will be a texture applied to a mesh. This mesh is transformed during the game to give the appearance of the board moving forward at varying speeds.

The sound

Lastly, the game should have some sort of audible feedback to add to the impression that the longboard is moving along the ground. A "clacking wheel" sound will be loaded into the game and its pitch adjusted based on the speed of the longboard or the sound stopped if the longboard isn't moving.

Initializing up the game

3

The foundational class in the GamePlay library is, not too surprisingly, the `Game` class. It includes the framework code you need to do many things that are common in games, such as starting and stopping the game, rendering the game frames, and maintaining basic game state. So, to create a game using GamePlay, you start by creating a class that inherits from `Game`.

For example, for the Longboard game, we create the following class:

```
class LongboardGame : public Game
```

Next, you'll almost always need to override the following methods:

```
void initialize();
void update();
void render();
void finalize();
```

There are a bunch of tasks you have to perform once and only once before your game starts. In GamePlay, you look after those tasks in the `initialize()` method. For the Longboard sample, a set of properties for the default render state is created using the `RenderState::StateBlock` class and this will be used throughout the sample. We also calculate our view/projection matrix, load some game entities, and set initial physics parameters.

Here's the code:

```
void LongboardGame::initialize()
{
    // Create our render state block that will be reused across all materials
    _stateBlock = RenderState::StateBlock::create();
    _stateBlock->setCullFace(true);
    _stateBlock->setBlend(true);
    _stateBlock->setBlendSrc(RenderState::BLEND_SRC_ALPHA);
    _stateBlock->setBlendDst(RenderState::BLEND_ONE_MINUS_SRC_ALPHA);

    // Calculate initial matrices
    Matrix::createPerspective(45.0f, (float)getWidth() / (float)getHeight(),
        0.25f, 100.0f, &_projectionMatrix);
    Matrix::createLookAt(0, 1.75f, 1.35f, 0, 0, -0.15f, 0, 0.20f,
        -0.80f, &_viewMatrix);
    Matrix::multiply(_projectionMatrix, _viewMatrix, &_viewProjectionMatrix);

    // Build game entities
    buildGround();
    buildBoard();
    buildWheels();
    buildGradient();

    // Set initial board physics
    _direction.set(0, 0, -1);
    _velocity = VELOCITY_MIN_MS;
}
```

A lot of the initialization work is hidden behind the build methods that create the game entities. It's also in those methods where you'll see more of the power of the GamePlay library.

The game entities (things that appear in the game) are built using OpenGL ES 2.0 under the covers. We won't cover OpenGL ES in this tutorial, but you can check out the following resources on the web for more information:

- OpenGL ES 2.0 Reference Pages
- OpenGL ES 2.0 Programming Guide
- OpenGL ES 2_X - The Standard for Embedded Accelerated 3D Graphics

What we will cover is the interface the GamePlay library provides on top of OpenGL ES. The geometry of a 3-D object in OpenGL is specified as a set of triangles. That set of triangles, taken together, are called a mesh. Some other names for a mesh are a model, an entity, or an object.

GamePlay includes a `Mesh` class that has a number of methods that make it easier to manage meshes. One such method, `createQuad()`, lets you create a quadrilateral (four sided mesh or a plane) with a single call, which saves you from building it out of two triangles. That call is used to build the mesh that represents all the objects in our longboard sample (board, wheels, ground, and gradient).

For example, here's the code for `buildGround()`. Notice that GamePlay provides a `Vector3` support class. Here, `WORLD_SIZE` is just a constant that specifies - surprise, surprise - the extent of the game world.

```
void LongboardGame::buildGround()
{
     Mesh* groundMesh = Mesh::createQuad(Vector3(-WORLD_SIZE, 0, -WORLD_SIZE),
                                         Vector3(-WORLD_SIZE, 0, WORLD_SIZE),
                                         Vector3(WORLD_SIZE, 0, -WORLD_SIZE),
                                         Vector3(WORLD_SIZE, 0, WORLD_SIZE) );
}
```

That looks after the basic geometry. Next, we have to specify the material that the ground is actually made of. In OpenGL, a material is really a statement of how an object interacts with light. GamePlay lets you create a material by either specifying an effect (using the `Effect` class) or by specifying a vertex and fragment shader program. In this sample, we specify shader programs.

```
_ground = Model::create(groundMesh);

Material* groundMaterial =
    Material::createMaterial("res/shaders/textured.vsh","res/shaders/textured.fsh");

groundMaterial->setStateBlock(_stateBlock);
```

An OpenGL vertex shader is a set of instructions, written in the GL shader language (GLSL), that describes how to transform vertex data during the rendering process. For example, you can alter the positions of vertices so that the shape they define appears to flutter in the wind. You can manipulate depth and color information in the vertex shader.

The following code implements the vertex shader used with the ground material (textured.vsh).

```
uniform mat4 worldViewProjection;

attribute vec3 a_position;
attribute vec3 a_normal;
attribute vec2 a_texCoord;

varying vec3 vnormal;
varying vec2 vtexCoord;

void main()
{
    gl_Position = worldViewProjection * vec4(a_position, 1);
    vnormal = a_normal;
    vtexCoord = a_texCoord;
}
```

Next, we specify the fragment shader. A fragment shader is like a vertex shader except that it manipulates sets of pixels and their properties instead of vertex data. You can, for example, use fragment shaders to cause blur effects or sepia tone effects.

```
precision highp float;

varying vec3 vnormal;
varying vec2 vtexCoord;

uniform sampler2D texture;
uniform vec2 textureRepeat;
uniform vec2 textureTransform;

void main()
{
    gl_FragColor = texture2D(texture, vtexCoord * textureRepeat + textureTransform);
}
```

We also set the render state of the ground material using the `setStateBlock()` function. After specifying the material of the ground, we specify its texture. GamePlay provides a `Texture` class that is designed to make this really easy. We just construct a `Texture` object, `groundTexture`, by specifying the location of a PNG file that we want to apply to the mesh. The second parameter, a Boolean, indicates whether or not mipmaps should be generated. Mipmaps are precalculated, resized versions of the original texture image. Using mipmapping speeds up rendering and reduces artifacts caused by anti-aliasing.

```
Texture::Sampler* groundSampler = groundMaterial->
    getParameter("texture")->setValue("res/textures/tileable_asphalt.png", true);
```

Next, we set the wrap mode of the texture. The first `true` parameter specifies that the texture should repeat horizontally, the second `true` parameter says to also repeat vertically. This allows the ground texture to fill the screen.

```
groundSampler->setWrapMode(Texture::REPEAT, Texture::REPEAT);
```

Next, we set other texture characteristics for the material that we just created.

```
groundMaterial->getParameter("worldViewProjection")-
>setValue(&_groundWorldViewProjectionMatrix);
groundMaterial->getParameter("textureRepeat")->setValue(Vector2(WORLD_SIZE/2,
WORLD_SIZE/2));
groundMaterial->getParameter("textureTransform")->setValue(&_groundUVTransform);
```

Lastly, we release all of the objects that are owned by mesh instances.

```
SAFE_RELEASE(groundMesh);
```

This setup is repeated for the board, wheels, and gradient.

Updating the game

4

A key part of this sample and any game or animation is gathering user input and updating the state of the game so that the input is reflected when the game world is rendered.

So, in the overridden `update()` method, you have to gather input and update game state. For the Longboard sample, you need to know how the user has moved the tablet. GamePlay uses the terms pitch and roll, which are commonly used when describing the motion of an aircraft, for the two motions we need to capture.

In GamePlay, you can get the pitch and roll with a single method call: `getAccelerometerPitchAndRoll(&pitch,&roll)`. In the code snippet below, the pitch and roll are read and clamped so they fall between a preset maximum and minimum value.

```
void LongboardGame::update(long elapsedTime)
{
    // Query the accelerometer
    float pitch, roll;
    Input::getAccelerometerPitchAndRoll(&pitch, &roll);

    // Clamp angles
    pitch = fmax(fmin(pitch, PITCH_MAX), PITCH_MIN);
    roll = fmax(fmin(roll, ROLL_MAX), -ROLL_MAX);
}
```

Next, the throttle is calculated based on the pitch angle of the tablet. You can think of the throttle value as representing how far down the gas pedal is pressed. The sound of the clacking wheels is then adjusted to match the throttle and the velocity of the board is updated accordingly. Notice that we're using a really simple formula for velocity here that ignores acceleration. In games, you don't have to apply real physics; sometimes it's just not necessary. In fact, in some cases making the physics too realistic can ruin the playability of a game.

```
float throttle = 1.0f - ((pitch - MIN_PITCH) / PITCH_RANGE);

if (throttle > 0.0f)
{
    if (_wheelsSound->getState() != AudioSource::PLAYING)
        _wheelsSound->play();

        _wheelsSound->setPitch(throttle);
}
else
{
    _wheelsSound->stop();
}

_velocity = VELOCITY_MIN_MS + ((VELOCITY_MAX_MS - VELOCITY_MIN_MS) * throttle);
```

To look after direction, we create a `Matrix` object. We then use the `createRotationY()` method to populate the matrix so that it represents a rotation about the y-axis. The calculation takes into account the roll input value that we retrieved above and the maximum turn rate. The direction of the longboard is stored in a `Vector3` object and was initialized to (0,0,1). So, we apply the rotation matrix to the direction vector and then renormalize it. The result is a unit vector that points in the new direction.

```
static Matrix rotMat;
 Matrix::createRotationY(-MATH_DEG_TO_RAD((TURN_RATE_MAX_MS * elapsedTime) *
    (roll / ROLL_MAX) * throttle), &rotMat);
 rotMat.transformVector(&_direction);
 _direction.normalize();
```

Next we transform the ground, the longboard wheels, and the board itself (including tilting it appropriately). Again, we use the matrices and matrix operations provided by the GamePlay library. In this case, that includes some perspective matrices. Note that we don't actually move the board itself, but instead we move the ground under it to simulate the board moving.

```
 Matrix::multiply(rotMat, _groundWorldMatrix, &_groundWorldMatrix);
 Matrix::multiply(_viewProjectionMatrix, _groundWorldMatrix,
&_groundWorldViewProjectionMatrix);


 Matrix::createScale(1.2f, 1.2f, 1.2f, &_wheelsWorldMatrix);
 _wheelsWorldMatrix.translate(-roll / ROLL_MAX * 0.05f, 0, 0.05f);
 _wheelsWorldMatrix.rotateY(MATH_DEG_TO_RAD(roll * 0.45f));
 Matrix::multiply(_viewProjectionMatrix, _wheelsWorldMatrix,
&_wheelsWorldViewProjectionMatrix);


 Matrix::createScale(1.25f, 1.25f, 1.25f, &_boardWorldMatrix);
 _boardWorldMatrix.translate(0, 0, 0.65f);
 _boardWorldMatrix.rotateZ(MATH_DEG_TO_RAD(roll * 0.5f));
 _boardWorldMatrix.rotateY(MATH_DEG_TO_RAD(roll * 0.1f));
 Matrix::multiply(_viewProjectionMatrix, _boardWorldMatrix,
&_boardWorldViewProjectionMatrix);
```

Using the following code, we make it look like the board is moving by transforming the ground's texture coordinates.

```
_groundUVTransform.x += -_direction.x * (_velocity * elapsedTime);
 _groundUVTransform.y += -_direction.z * (_velocity * elapsedTime);
 if (_groundUVTransform.x >= 1.0f)
 {
     _groundUVTransform.x = 1.0f - _groundUVTransform.x;
 }
 if (_groundUVTransform.y >= 1.0f)
 {
     _groundUVTransform.y = 1.0f - _groundUVTransform.y;
 }
```

Rendering the game

The `render()` method for our sample is just a few lines of code. We need all of our game entities to render —or draw— themselves on the screen. So, we clear the color and depth buffers and call each game entity's `draw()` method.

```
void LongboardGame::render()
{
    // Clear the color and depth buffers.
    clear(CLEAR_COLOR, Vector4::one(), 1.0f, 0);

    // Draw the scene
    _ground->draw();
    _wheels->draw();
    _board->draw();
    _gradient->draw();
}
```

Again, GamePlay abstracts away a complex process - that of rendering a game entity in this case. Under the covers, these objects are represented by instances of the `MeshInstance` class.

So that's it, the game has been set up, updated, and rendered and is now ready to launch and play!

Try it yourself

The GamePlay library is designed to make it really easy to create simple games. After you understand how the Longboard sample is put together, try changing some things. Start with simple stuff, such as the texture files or some of the constants. When you really have the hang of things, try creating your own game from scratch using Longboard nearby as a cheat sheet. Have fun!

Legal notice

6

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.

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