Introduction

The past tutorials, apart from designing and writing a simple space invaders game, have been building an infrastructure for comparing the use of rendering techniques. So, hopefully this is going to pay off now when we attempt to start using LWJGL. So in this tutorial we're going to cover:

- Initialising LWJGL
- Rendering sprites in LWJGL
- Keyboard input in LWJGL

This is going to a relatively short tutorial for several reasons. LWJGL is very simple to work with and has some strong similarities to JOGL and hence the last tutorial.

The final game can be see **here**. The complete source for the tutorial can be found **here**. Its intended that you read through this tutorial with the source code at your side. The tutorial isn't going to cover every line of code but should give you enough to fully understand how it works.

Context hightlighted source is also available here:

- **LWJGLGameWindow.java**
- **LWJGLSprite.java**
- **TextureLoader.java**
- **Texture.java**
- **ResourceFactory.java**

Disclaimer: This tutorial is provided as is. I don't guarantee that the provided source is perfect or that that it provides best practices.

LWJGL

So whats LWJGL then? Well, LWJGL is the Lightweight Java Games Library. Its lightweight because it doesn't rely on much of the internals of Java. This is good because it allows you to keep the final distributable size down (by cutting out the bits you don't need). Java is a no brainer, its written and designed to work with Java. Most importantly its a game library, not just a binding to OpenGL.
This seems like a trivial point, especially when being illustrated by such a simplistic demo as this spaceinvaders game. However, LWJGL provides not just a graphics binding but a binding to OpenAL (an open standards sound system). In addition, LWJGL gives your access to input devices. This means your spaceinvaders game could be controlled from a gamepad and the aliens could squeltch as they die, all from one library. Worth thinking about.

Adding a new rendering implementation for our SpaceInvaders game should be pretty simple (especially as in this case I didn't have to write the code, thanks Matzon). We're going to add 4 classes then plug then into our architecture.

LWJGL Window

Initialisation

As with the other rendering layers the first thing we need to do is create a window and notify the infrastructure. The initial part of understanding LWJGL is to realise there is only a single window, the one in which the game runs. So, to create and configure our game window we need to configure a display mode for that window like this:

```
/**
 * Sets the display mode for fullscreen mode
 */
private boolean setDisplayMode() {
    try {
        // get modes
        DisplayMode[] dm = org.lwjgl.util.Display.getAvailableDisplayModes(width,
                                                    height, -1, -1, -1, 60, 60);

        org.lwjgl.util.Display.setDisplayMode(dm, new String[] {
             "width=" + width,
             "height=" + height,
             "freq=" + 60,
             "bpp=" + org.lwjgl.opengl.Display.getDisplayMode().getBitsPerPixel()
        });

        return true;
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Unable to enter fullscreen, continuing in windowed mode");
    }

    return false;
}
```

Since there is only one window, the static call on the "Display" class makes sense. The next thing to accept is that there is only one OpenGL context at any given time. Hence we can also access this in a purely static way. These sort of assumptions are one of the things that make LWJGL simple to use.

As you can see from the following code, OpenGL calls are simply prefixed with the class "GL11":

```
public void startRendering() {
        try {
                setDisplayMode();
                Display.create();

                // grab the mouse, dont want that hideous cursor when we're playing!
                Mouse.setGrabbed(true);

                // enable textures since we're going to use these for our sprites
                GL11.glEnable(GL11.GL_TEXTURE_2D);

                // disable the OpenGL depth test since we're rendering 2D graphics
                GL11.glDisable(GL11.GL_DEPTH_TEST);

                GL11.glMatrixMode(GL11.GL_PROJECTION);
                GL11.glLoadIdentity();

                GL11.glOrtho(0, width, height, 0, -1, 1);

                textureLoader = new TextureLoader();

                if(callback != null) {
                        callback.initialise();
                }
        } catch (LWJGLException le) {
                callback.windowClosed();
        }

        gameLoop();
}
```

So, we create our window, initialise OpenGL by setting our options and using glOrtho() to configure our view (for more on this see SpaceInvaders 103). Next we create a texture loader, which we will use to load our sprite images. Finally we notify the infrastructure that the initialisation has been completed.

Also note the line "Mouse.setGrabbed(true);". As we'll see later, the mouse and keyboard are also monitored from static contexts. This line simple says that the mouse cursor shouldn't be displayed in the window.

Game Loop

Next lets look at the game loop for LWJGL. As you can see from the code below its very similar to the JOGL version in the last tutorial. Essentially we clear the screen, tell the infrastructure to do any rendering it needs to do, and finally update the windows contents.

```
private void gameLoop() {
        while (gameRunning) {
                // clear screen
                GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
                GL11.glMatrixMode(GL11.GL_MODELVIEW);
                GL11.glLoadIdentity();

                // let subsystem paint
                if (callback != null) {
                        callback.frameRendering();
                }

                // update window contents
                Display.update();

                if(Display.isCloseRequested() || Keyboard.isKeyDown(Keyboard.KEY_ESCAPE)) {
                        gameRunning = false;
                        Display.destroy();
                        callback.windowClosed();
                }
        }
}
```

The final addition here is to check our exit cases. If the user has closed the window (Window.isCloseRequested()) or if the escape key has been pressed (Keyboard.isKeyDown(Keyboard.KEY_SCAPE)) then we need to notify the infrastructure and destroy the game window. In LWJGL we accomplish this by calling Window.destroy().

Keyboard Handling

LWJGL provides its own keyboard handling object. This means that we don't need to use AWT events to catch key pressed we can simply poll for them using LWJGL's Keyboard class. So the implementation of check for key presses looks like this:

```
public boolean isKeyPressed(int keyCode) {
        // apparently, someone at decided not to use standard
        // keycode, so we have to map them over:
        switch(keyCode) {
        case KeyEvent.VK_SPACE:
                keyCode = Keyboard.KEY_SPACE;
                break;
        case KeyEvent.VK_LEFT:
                keyCode = Keyboard.KEY_LEFT;
                break;
        case KeyEvent.VK_RIGHT:
                keyCode = Keyboard.KEY_RIGHT;
```

```
                break;
        }

        return Keyboard.isKeyDown(keyCode);
    }
```

The switch() here is a little confusing, why do we need to map VK_SPACE to KEY_SPACE and so on? LWJGL has stuck to the original key codes provided by "most" operating systems and since its a games oriented API this makes perfect sense. Java's AWT however uses its own set of key codes, which means we need a mapping between the two sets. As you can see, for the purposes of SpaceInvaders this is very simple.

LWJGL Sprite

Since LWJGL allows to access OpenGL the code for drawing a sprite on the screen is almost identical to the JOGL version (see part 3). Essentially we draw a rectangle at the right location and at the right size on the screen. We map our sprite texture across this rectangle. The code looks like this:

```
    public void draw(int x, int y) {
        // store the current model matrix
        GL11.glPushMatrix();

        // bind to the appropriate texture for this sprite
        texture.bind();

        // translate to the right location and prepare to draw
        GL11.glTranslatef(x, y, 0);
        GL11.glColor3f(1,1,1);

        // draw a quad textured to match the sprite
        GL11.glBegin(GL11.GL_QUADS);
        {
            GL11.glTexCoord2f(0, 0);
            GL11.glVertex2f(0, 0);
            GL11.glTexCoord2f(0, texture.getHeight());
            GL11.glVertex2f(0, height);
            GL11.glTexCoord2f(texture.getWidth(), texture.getHeight());
            GL11.glVertex2f(width,height);
            GL11.glTexCoord2f(texture.getWidth(), 0);
            GL11.glVertex2f(width,0);
        }
        GL11.glEnd();


        // restore the model view matrix to prevent contamination
        GL11.glPopMatrix();
    }
```

Note that the only significant change is the use of our GL context. In LWJGL instead of have a GL class implementation, we have static access to the GL11 class which provides us our OpenGL access.

Textures

Like most of the code we've seen here, the texture loader is very similar to the JOGL version. To understand what the code is doing refer to SpaceInvaders 103. There is no significant difference between creating and manipulating textures in JOGL or LWJGL.

Plugging it Together - ResourceFactory updates

Our final step is to plug our new rendering layer into the existing SpaceInvaders infrastructure. To do this we need to make two changes in our ResourceFactory class. The first issue is that in our new rendering layer we use a different game window. Here's the update:

```
    public GameWindow getGameWindow() {
        // if we've yet to create the game window, create the appropriate one
        // now
        if (window == null) {
            switch (renderingType) {
                case JAVA2D:
                {
                    window = new Java2DGameWindow();
                    break;
                }
                case OPENGL_JOGL:
                {
                    window = new JoglGameWindow();
                    break;
                }
                case OPENGL_LWJGL:
                {
                    window = new LWJGLGameWindow();
                    break;
                }
            }
        }

        return window;
    }
```

All we had to do was add a new leg to the switch will return the appropriate game window for LWJGL. Finally, we need to make

a similar change for sprites. Here we're going to create an LWJGLSprite if we're using LWJGL to render:

```
public Sprite getSprite(String ref) {
        if (window == null) {
                throw new RuntimeException("Attempt to retrieve sprite before game window was created");
        }

        switch (renderingType) {
                case JAVA2D:
                {
                        return Java2DSpriteStore.get().getSprite((Java2DGameWindow) window,ref);
                }
                case OPENGL_JOGL:
                {
                        return new JoglSprite((JoglGameWindow) window,ref);
                }
                case OPENGL_LWJGL:
                {
                        return new LWJGLSprite((LWJGLGameWindow) window,ref);
                }
        }

        throw new RuntimeException("Unknown rendering type: "+renderingType);
}
```

And thats it! As noted this is relatively short tutorial. This is partly because our existing infrastruture was designed specifically to support showing the changes in rendering layers. However, hopefully this tutorial has also shown has simple it is to use LWJGL to develop 2D games.

Exercises for the Reader

The following extensions of the game may help in understanding how game hangs together.

Add sound effects using LWJGL

As mentioned above, LWJGL has support for many aspects of games development. One of its strong suits is sound and music using OpenAL. Add some simple sound effects for when the player shoots and the aliens are destroyed. This should help in understanding where to look for information on LWJGL.

Add controller support using LWJGL

LWJGL also has an API for reading input from game controllers connected to your machine. If you have a game controller handy adding support for it often changes the way a game feels altogether. Again, adding this sort of functionality will help learning where to look for more information on LWJGL.

Support Fullscreen mode in LWJGL

Full screen mode is actually more normal for games development and hence for LWJGL. A simple change to the create() call should change the game into a fullscreen application. However, many systems don't support this functionality fully. Its worth experimenting with this addition with a series of different systems.

Further Reading

I'd recommend you take a look at the following material, it'll help support taking the current code to the next stage.

- **The Home of LWJGL**

Credits

Tutorial and Base Source written by **Kevin Glass**
LWJGL Source written by Brian Matzon
Game sprites provided by **Ari Feldman**
A large number of people over at the **Java Gaming Forums**

---

previous **up** **next**

Space Invaders 103 - Refactoring and OpenGL Asteroids - 3D Rendering in Java

printer-friendly version | **add new comment**