Introduction

This tutorial follows on from **Space Invaders 102**. The initial aim of this tutorial is to understand rendering 2D graphics using JOGL and compare this to Java2D. However, to do this clearly the tutorial attempts to refactor the original space invaders source code to pull out exactly the differences between the two methods of rendering. There are parts of the tutorial that can be ignored if the reader isn't interested in the design side of development and these will be noted throughout.
The result of this tutorial can be seen **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 highlighted source is also available here:

Game.java
GameWindow.java
GameWindowCallback.java
ResourceFactory.java
Sprite.java
Entity.java
ShipEntity.java
ShotEntity.java
AlienEntity.java
SystemTimer.java
Java2DGameWindow.java
Java2DSprite.java
Java2DSpriteStore.java
JoglGameWindow.java
JoglSprite.java
Texture.java
TextureLoader.java
Keyboard.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.

Refactoring Space Invaders

In this section we're going to look at what "refactoring" is and why we're doing it in this case. If you're not interested in design just skip it.
Our last space invaders development was solely based around Java2D. This time round we'd like to support OpenGL. Now we could just rehack the source code to replace the Java2D implementation with an OpenGL version. While in many cases this would be perfectly logical, in this case what'd we'd really like is to be able to compare and contrast the two versions. Even more importantly, in the future we could anticipate using yet another different method of rendering. This is where refactoring comes in.

Refactoring is generally the process of taking a piece of code that put together to a specific job and redesigning it to be more flexible and/or easier to maintain. It happens to most developers at some point, when starting a project the intentions were one thing. Later on, having understood the problem more fully and having realised the mistakes it seems like a good idea to start a fresh piece of code or to rewrite what you have. At this point the best bet is to refactor the code, normally before adding anything else. In this way, often refactoring itself can be a very thankless process since the functionality you have afterwards should be very similar to what you had before. However, as a good Java/OO developer you have to trust that redesigning your source is going to save you time in the long run.

In games its generally desirable to have decided what you're going to use to render at the begining. This means you don't spend alot of time trying to get multiple rendering methods displaying something that is very easy to produce in one of them. For instance, fast alpha blending is difficult to achieve in Java2D, however in OpenGL its very easy. Enforcing that we support both rendering methods can be difficult and apart from anything else a waste of valuable development time. All this being said, in this case we're trying look at the differences in rendering and so thats exactly what we're going to do. We're going introduce a set of interfaces that describe what we need our rendering layer to be able to do for us. Then we'll supply implementations of these interfaces for each of our intended rendering layers.

Rendering

So first off we need to consider what we need to render. Essentially we need a window to draw our game in and a loop to actively keep rendering our game. Here we go then, an interface describing what we need from our window.

```
package org.newdawn.spaceinvaders;

/**
 * The window in which the game will be displayed. This interface exposes just
 * enough to allow the game logic to interact with, while still maintaining an
 * abstraction away from any physical implementation of windowing (i.e. AWT, LWJGL)
 *
 * @author Kevin Glass
 */
public interface GameWindow {

        /**
         * Set the title of the game window
         *
         * @param title The new title for the game window
         */
        public void setTitle(String title);

        /**
         * Set the game display resolution
         *
         * @param x The new x resolution of the display
         * @param y The new y resolution of the display
```

```
         */
        public void setResolution(int x,int y);

        /**
         * Start the game window rendering the display
         */
        public void startRendering();

        /**
         * Set the callback that should be notified of the window
         * events.
         *
         * @param callback The callback that should be notified of game
         * window events.
         */
        public void setGameWindowCallback(GameWindowCallback callback);

        /**
         * Check if a particular key is pressed
         *
         * @param keyCode The code associate with the key to check
         * @return True if the particular key is pressed
         */
        public boolean isKeyPressed(int keyCode);
}
```

setTitle(), setResolution() and startRendering() are directly related to the window and game loop. We have two extra additions here. First off, just like rendering to the screen we need our game window to provide with a way to check if a given key is pressed. In the last version we used standard KeyListeners to do this for us, however if we want to be truely independant of how our game is rendered we need this to be abstract this as well. In addition we've added setGameWindowCallback(), this is how we're going to hook into the game loop. The class that we set as GameWindowCallback is going to be notified each time the frame is being rendered. This will give us a chance to draw our game in frame. The GameWindowCallback interface looks like this:

```
package org.newdawn.spaceinvaders;

/**
 * An interface describing any class that wishes to be notified
 * as the game window renders.
 *
 * @author Kevin Glass
 */
public interface GameWindowCallback {
        /**
         * Notification that game should initialise any resources it
         * needs to use. This includes loading sprites.
         */
        public void initialise();

        /**
         * Notification that the display is being rendered. The implementor
         * should render the scene and update any game logic
         */
        public void frameRendering();

        /**
         * Notification that game window has been closed.
         */
        public void windowClosed();
}
```

As you can see the GameWindowCallback has just a bit more than frame rendering notification in it. We have an initialise method, thats going to be called at startup. Many resources that we load are dependant on the rendering layer being in the right state. For instance, in Java2D when we load our sprites we need to make sure we've changed graphics modes first, so the sprites get loaded in the right format. Finally theres an additional hook to notify when the game rendering window has been closed.

So, our game window implementations will be responsible for creating an actual window, and running a game loop. In this game loop the GameWindow is going to call frameRendering() to let us know that we need to draw our game, but how are we going to draw our game without knowing which rendering layer is in use? To solve this we're going to have to create an interface for our main drawing tool, Sprite. Intead of Sprite being a concrete class that draws a sprite to Java2D we're going to make it an interface that can be implemented by the different rendering layers. Here's what we required from the sprite:

```
package org.newdawn.spaceinvaders;

/**
 * A sprite to be displayed on the screen. Note that a sprite
 * contains no state information, i.e. its just the image and
 * not the location. This allows us to use a single sprite in
 * lots of different places without having to store multiple
 * copies of the image.
 *
 * @author Kevin Glass
 */
public interface Sprite {


        /**
```

```
         * Get the width of the drawn sprite
         *
         * @return The width in pixels of this sprite
         */
        public int getWidth();

        /**
         * Get the height of the drawn sprite
         *
         * @return The height in pixels of this sprite
         */
        public int getHeight();

        /**
         * Draw the sprite onto the graphics context provided
         *
         * @param x The x location at which to draw the sprite
         * @param y The y location at which to draw the sprite
         */
        public void draw(int x,int y);
}
```

For those who followed the first two tutorials this interface should look very familiar. Essentially it matches the interface of the original Sprite class in the Java2D implementations. This happens quite often when refactoring code, its normally called "extracting the interface". What we're trying to say here is that any class that represents a sprite that can be drawn to the screen much be able to do the specified things.

Now we've defined what resources we need to play out game, GameWindow and Sprite, and we've defined what these resources must be able to do for us. Next we can look at how we implement these two resources in the different rendering methods.

Refactorying Java2D Rendering

Since we've already implemented the game in Java2D we should have all the elements we need already coded. We now need to rearrange the old implementation to match up with our interfaces. Lets look at Java2DGameWindow first, this will be the class that implements GameWindow and maintains a normal AWT window. Essentially this class will create a JFrame and maintain a game loop like our old Game class did.
The only significant change is in the game loop, instead of drawing the sprites within the actual loop we're going to call a method on the GameWindowCallback and rely on which ever class implements that interface to draw our sprites. The game loop looks like this:

```
private void gameLoop() {
        while (gameRunning) {
                // Get hold of a graphics context for the accelerated
                // surface and blank it out
                g = (Graphics2D) strategy.getDrawGraphics();
                g.setColor(Color.black);
                g.fillRect(0,0,800,600);

                if (callback != null) {
                        callback.frameRendering();
                }

                // finally, we've completed drawing so clear up the graphics
                // and flip the buffer over
                g.dispose();
                strategy.show();
        }
}
```

We've extracted a portion of the original Game class which was responsible for setting up the window to be rendered in. Next, if a callback has been registered we notify it that a game frame is being rendered and hence it should draw anything to the screen its going to. Finally, another part of the original class that is responsible for swapping over our Java2D buffer strategy.

Everything else in Java2DGameWindow has been extracted from the original Game class with the exception of the keyboard handling. Since we're going to want similar keyboard handling in both OpenGL and Java2D this has been pulled out into a seperate class called "Keyboard". However, this will be covered further on in this tutorial.

The next thing we should look at is Java2DSprite. If you look through the code you'll find its almost identical to the original Sprite class from the Java2D tutorial. However, there is one small, but very important change, here:

```
public void draw(int x,int y) {
        window.getDrawGraphics().drawImage(image,x,y,null);
}
```

Note that the graphics context is no longer passed into the draw method. When we ask a Sprite to draw itself we do so from a class that does not know what type of rendering is going on. However, since a sprite must be being drawn into a window the sprite can now obtain its graphics context from the window itself (which is parameter to its construction).

Thats pretty much it for refactoring. We've made our original code much more abstract by introducing interfaces for each of the resources. In addition we've changed our Java2D code to match up with our new interfaces. Next lets look at the keyboard handling.

Implementing AWT Keyboard Input

In the first two tutorials we were very explicit about which keys we expected to`be pressed. While this suited our purposes for the first tutorial its not very reusable. It would be nice to have a central location to be able to check which keys are pressed. It would`also be nice to be able to hide all the tricky bits of keyboard manipulation away somewhere. Our GameWindow interface already has a isKeyPressed() method so all we need`now is one central class

to provide the checks.

```
public class Keyboard {
        /** The status of the keys on the keyboard */
    private static boolean[] keys = new boolean[1024];

    /**
     * Initialise the central keyboard handler
     */
    public static void init() {
        Toolkit.getDefaultToolkit().addAWTEventListener(new KeyHandler(),
                                            AWTEvent.KEY_EVENT_MASK);
    }

    /**
     * Initialise the central keyboard handler
     *
     * @param c The component that we will listen to
     */
    public static void init(Component c) {
        c.addKeyListener(new KeyHandler());
    }

    /**
     * Check if a specified key is pressed
     *
     * @param key The code of the key to check (defined in KeyEvent)
     * @return True if the key is pressed
     */
    public static boolean isPressed(int key) {
        return keys[key];
    }

    /**
     * Set the status of the key
     *
     * @param key The code of the key to set
     * @param pressed The new status of the key
     */
    public static void setPressed(int key,boolean pressed) {
        keys[key] = pressed;
    }
```

We are trying to write a nice reusable class so there will be some extra functionality in the eyboard class`that isn't used directly here. Its just nice to implement these things up front so there ready and waiting when you get round to needing them.

First in our Keyboard class we need a set of flags to indicate whether a given key is pressed and a couple of accessor methods to allow us to check if a key is pressed and indicate that a key is pressed. When we initialise the Keyboard class we can do it two ways. First, and most commonly`we can pass in a component. A key listener will be added to this component and the state recorded. On the other hand it sometimes helps to be able to respond to key presses on a global scale, not localised to a particular component. With this in mind theres a second init method which allows us to add a AWTEventListener directly to the AWT event queue. This will pick up any keyboard events at a global level.

Now on to the all important key listener class. This is a bit specialised so don't be surprised if it looks a bit complicated,

```
private static class KeyHandler extends KeyAdapter implements AWTEventListener {
        /**
         * Notification of a key press
         *
         * @param e The event details
         */
        public void keyPressed(KeyEvent e) {
                if (e.isConsumed()) {
                        return;
                }
            keys[e.getKeyCode()] = true;
        }

      /**
       * Notification of a key release
       *
       * @param e The event details
       */
      public void keyReleased(KeyEvent e) {
        if (e.isConsumed()) {
           return;
        }

        KeyEvent nextPress = (KeyEvent) Toolkit.getDefaultToolkit().
                                        getSystemEventQueue().
                                        peekEvent(KeyEvent.KEY_PRESSED);

        if ((nextPress == null) ||
              (nextPress.getWhen() != e.getWhen()) ||
              (nextPress.getKeyCode() != e.getKeyCode())) {
                        keys[e.getKeyCode()] = false;
         }
       }
```

```
        /**
         * Notification that an event has occured in the AWT event
         * system
         *
         * @param e The event details
         */
        public void eventDispatched(AWTEvent e) {
                if (e.getID() == KeyEvent.KEY_PRESSED) {
                keyPressed((KeyEvent) e);
                }
                if (e.getID() == KeyEvent.KEY_RELEASED) {
                keyReleased((KeyEvent) e);
            }
        }
    }
```

The first thing thats complicated about this class is that its actually being two different things. First of add its a key listener (well its a KeyAdapter) and secondly its implementing the AWTEventListener interface.

AWTEventListener allows us to be notified of events directly as they become available to the AWT event queue. While this is immensely useful, its also a bit dirty in that we get passed the events as **AWTEvent**. In our case we're only interested in key events so we can check the ID of the event and then palm off the event to the method that would normally recieve it. This all happens in eventDispatched().

The other part that makes this key handler difficult to understand is the extra bits of code in the keyReleased() method. It would seem to be as simple as indicating that the key is no longer pressed by setting the appropriate index in the keys array to false. However, theres an issue here where Java isn't quite as platform independant as we'd like it to be. When you hold a key down on a keyboard you get a slight pause then the key begins to repeat across the screen. This is normal everywhere. However, the notification for this in Java changes from platform to platform. On Windows when the key begins to repeat no extra notifications are sent, the key is considered still to be pressed. However, on Linux when a key begins to repeat a notifcation of release and then pressed is sent for each repeat. This means that if the user was to hold down the key then our little space ship would move a bit, then stop, then move a bit, then stop, and so on.

Getting round this is a little more complicated than you'd think. In this case we end up with the following code:

```
KeyEvent nextPress = (KeyEvent) Toolkit.getDefaultToolkit().
                                    getSystemEventQueue().
                                    peekEvent(KeyEvent.KEY_PRESSED);

if ((nextPress == null) ||
    (nextPress.getWhen() != e.getWhen()) ||
    (nextPress.getKeyCode() != e.getKeyCode())) {
        keys[e.getKeyCode()] = false;
}
```

When a keyReleased notification is recieved we search the pending AWT events by using peekEvent() for a keyPressed event. If there is a keyPressed event scheduled for the exact same moment the keyReleased event is scheduled then its auto-repeat on linux causing the events. At which point we ignore it. Complicated and annoying, Java has a few things like this that are slowly being sorted out.

Having gone through all that, as long as we've called one of the init methods on our Keyboard class we can now implement our GameWindow's isKeyPressed method by calling keyPressed() here. As we implement other rendering methods that use AWT (JOGL for instance) we can now reuse our class to handle key input.

Completing the Picture

The final step in our refactoring is to integrate our new resource interfaces with our game logic. After this we should be back to a working game, it won't look any different to before (apart from text is no longer supported so we'll use some fixed sprites intead). However, the new code structure is going to make it much easier to look at adding OpenGL.

Resource Factory

First off, how are we going to get hold of our new Java2D implementations of resources. We could just construct them directly but that makes us a bit too dependant on the rendering layer. Theres a fairly common pattern used for this type of problem called the "Factory". Essentially, we ask the factory for a particular type of resource. It returns up a concrete implementation as the interface we asked for. Sounds a bit strange, this is how it works out for instance in ResourceFactory for sprites:

```
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);
                }
        }

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

```
	}
```

We ask the ResourceFactory for a sprite given by a specific reference. Depending on the type of rendering we're using the appropriate type of sprite is created (be it Java2DSprite or JoglSprite). However, its passed back as the interface Sprite, this means that caller doesn't need to worry about which type of rendering is being used. Note, we use a JoglSprite here that we're going to go on an implement later in this tutorial.

If you check through the rest of ResourceFactory you'll find its pretty simple after understanding this. The only hook into the type of rendering performed is provided by the rather explcitly named "setRenderingType()". We'll call this from our main game class to set which type of rendering we want to use.

Game Logic Changes

Finally we need to update our main class to make use of all the new exciting bits of code we've added. After this is complete we can move on to adding a new rendering layer, i.e. OpenGL to our simple space invaders game.
The first thing to note is that since we've extracted all the window management and game loop functionality and added it to the Java2DGameWindow we can remove it all from the Game class. This leaves the game class as a collection of bits of game logic and a piece of code to render the game screen. In addition to removing the window management code we can also strip out the keyboard
code since that is also handled by the rendering layer.

Instead of creating the window directly we're going to ask the ResourceFactory class for it. We'll register ourselfs as the GameWindowCallback so we'll get notified of rendering. Finally, instead of starting out own game loop we simply ask the GameWindow to start rendering which is effectively our old game loop.

```
	public Game(int renderingType) {
		// create a window based on a chosen rendering method
		ResourceFactory.get().setRenderingType(renderingType);
		window = ResourceFactory.get().getGameWindow();

		window.setResolution(800,600);
		window.setGameWindowCallback(this);
		window.setTitle(windowTitle);
	}

	public void startGame() {
		window.startRendering();
	}
```

Now, we have a very pure and simple class. The final step in locking together the rendering layer and game logic is to modify the Game class to implement the GameWindowCallback interface. This means moving all resource loading into the initialise() method (note that this includes entity creation),
and modifying the gameLoop() method to be the frameRendering() method. We no longer need to manage the game loop since the GameWindow in use is going to do that for us.

Thats it, we're done. At this stage we should be able to play our Java2D version of the game as before. There are some minor changes in addition to those noted above which complete the picture which are to do with swapping over from explictly creating our resource to asking the ResourceFactory for them. Our next and probably more interesting step is going to be to provide an OpenGL rendering layer. We won't need to touch the game logic again so from here on should be plain sailing.

Implementing OpenGL Rendering

For those just interested in a guideline on how to use JOGL, here we go. We're now going to add a rendering layer using JOGL. However, it seems only fair to first consider alternatives for using OpenGL in Java.

OpenGL Choices

OpenGL is currently only accessable in Java through a native library (for more on these check out the discussion in **Space Invaders 102**. However, unlike timing, for OpenGL you're going to need a native library for any platform that you deploy on. This can increase your deployment size a great deal. In addition, if you rely on OpenGL you are also relying on the user having up-to-date drivers for their graphics card. However, this is a well known issue to most gamers and so doesn't normally cause any issue.
OpenGL does of course has many advantages. On most platforms you'll get hardware acceleration, there are many special effects that are easily achievable with OpenGL.

GL4Java

One of the original native bindings for Java. Currently this binding is no longer supported and has fallen behind the OpenGL standard. Perfomance is also poor in comparison to other possible APIs.

Lighweight Java Gaming Library (LWJGL)

This binding is far more than simply an access layer to OpenGL. It also contains a binding to OpenAL (the audio library) and some custom interfaces to input controllers. There is a large community of developers around the library and support is always forth coming.
The only downside of the LWJGL is that you need to buy into all of the ideas behind the library or none of them. Assuming you agree with everything the library has been written around then this isn't a problem.

Java OpenGL (JOGL)

JOGL is a simple OpenGL binding produced by the community but spearheaded by Sun. JOGL only exposes OpenGL and does so in a very clean and simple manner. However, it is dependant on AWT and as such isn't suitable for native compilation.
With luck JOGL will be making it into the JDK at some later date. As/when this happens JOGL can be considered part of pure Java. With this in mind I'll use JOGL as our native layer for this tutorial.

JOGL Interface

Next we'll look at producing a rendering layer for our space invaders game based on OpenGL. The first thing we need to do is implement a GameWindow for OpenGL. To do this we're going to implement the main interface that JOGL

provides, GLEventListener. This interface consists of four methods described below.
initialise() - Called by JOGL to request that you initialise any resource you require. This maps directly to our GameWindow's initialise() method. The GLDrawable passed in can be used to access GL during this method.

display() - Called by JOGL to request that you render the scene. This effectively our hook into JOGL for our game loop. The GLDrawable passed in can be used render to the screen while in this method.

reshape() - Called to indicate that the window has been resized or moved. Most of the time we don't need to do anything here apart from adapting for resolution.

displayChanged() - Called to indicate that the graphics mode has been changed. For our purposes this will never happen.

Here's how these things look in the our JoglGameWindow. We'll look at the rest of the code piece by piece later on.

```
public class JoglGameWindow implements GLEventListener,GameWindow {
        /** The frame containing the JOGL display */
        private Frame frame;
        /** The callback which should be notified of window events */
        private GameWindowCallback callback;
        /** The width of the game display area */
        private int width;
        /** The height of the game display area */
        private int height;
        /** The canvas which gives us access to OpenGL */
        private GLCanvas canvas;
        /** The OpenGL content, we use this to access all the OpenGL commands */
        private GL gl;
        /** The loader responsible for converting images into OpenGL textures */
        private TextureLoader textureLoader;

        /**
         * Create a new game window that will use OpenGL to
         * render our game.
         */
        public JoglGameWindow() {
                frame = new Frame();
        }

        /**
         * Retrieve access to the texture loader that converts images
         * into OpenGL textures. Note, this has been made package level
         * since only other parts of the JOGL implementations need to access
         * it.
         *
         * @return The texture loader that can be used to load images into
         * OpenGL textures.
         */
        TextureLoader getTextureLoader() {
                return textureLoader;
        }

        /**
         * Get access to the GL context that can be used in JOGL to
         * call OpenGL commands.
         *
         * @return The GL context which can be used for this window
         */
        GL getGL() {
                return gl;
        }

        /**
         * Set the title of this window.
         *
         * @param title The title to set on this window
         */
        public void setTitle(String title) {
                frame.setTitle(title);
        }

        /**
         * Set the resolution of the game display area.
         *
         * @param x The width of the game display area
         * @param y The height of the game display area
         */
        public void setResolution(int x, int y) {
                width = x;
                height = y;
        }

        /**
         * Start the rendering process. This method will cause the
         * display to redraw as fast as possible.
         */
        public void startRendering() {
                canvas = GLDrawableFactory.getFactory().createGLCanvas(new GLCapabilities());
                canvas.addGLEventListener(this);
                canvas.setNoAutoRedrawMode(true);
                canvas.setFocusable(true);
```

```
        Keyboard.init(canvas);

        Animator animator = new Animator(canvas);

        // Setup the canvas inside the main window
        frame.setLayout(new BorderLayout());
        frame.add(canvas);
        frame.setResizable(false);
        canvas.setSize(width, height);
        frame.pack();
        frame.show();

        // add a listener to respond to the user closing the window. If they
        // do we'd like to exit the game
        frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        if (callback != null) {
                                callback.windowClosed();
                        } else {
                                System.exit(0);
                        }
                }
        });

        // start a animating thread (provided by JOGL) to actively update
        // the canvas
        animator.start();
}

/**
 * Register a callback that will be notified of game window
 * events.
 *
 * @param callback The callback that should be notified of game
 * window events.
 */
public void setGameWindowCallback(GameWindowCallback callback) {
        this.callback = callback;
}

/**
 * Check if a particular key is current pressed.
 *
 * @param keyCode The code associated with the key to check
 * @return True if the specified key is pressed
 */
public boolean isKeyPressed(int keyCode) {
        return Keyboard.isPressed(keyCode);
}

/**
 * Called by the JOGL rendering process at initialisation. This method
 * is responsible for setting up the GL context.
 *
 * @param drawable The GL context which is being initialised
 */
public void init(GLDrawable drawable) {
        // get hold of the GL content
        gl = drawable.getGL();

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

        // set the background colour of the display to black
        gl.glClearColor(0, 0, 0, 0);
        // set the area being rendered
        gl.glViewport(0, 0, width, height);
        // disable the OpenGL depth test since we're rendering 2D graphics
        gl.glDisable(GL.GL_DEPTH_TEST);

        textureLoader = new TextureLoader(drawable.getGL());

        if (callback != null) {
                callback.initialise();
        }
}

/**
 * Called by the JOGL rendering process to display a frame. In this
 * case its responsible for blanking the display and then notifing
 * any registered callback that the screen requires rendering.
 *
 * @param drawable The GL context component being drawn
 */
public void display(GLDrawable drawable) {
        // get hold of the GL content
        gl = canvas.getGL();

        // clear the screen and setup for rendering
        gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
```

```
                gl.glMatrixMode(GL.GL_MODELVIEW);
                gl.glLoadIdentity();

                // if a callback has been registered notify it that the
                // screen is being rendered
                if (callback != null) {
                        callback.frameRendering();
                }

                // flush the graphics commands to the card
                gl.glFlush();
        }

        /**
         * Called by the JOGL rendering process if and when the display is
         * resized.
         *
         * @param drawable The GL content component being resized
         * @param x The new x location of the component
         * @param y The new y location of the component
         * @param width The width of the component
         * @param height The height of the component
         */
        public void reshape(GLDrawable drawable, int x, int y, int width, int height) {
                gl = canvas.getGL();

                // at reshape we're going to tell OPENGL that we'd like to
                // treat the screen on a pixel by pixel basis by telling
                // it to use Orthographic projection.
                gl.glMatrixMode(GL.GL_PROJECTION);
                gl.glLoadIdentity();

                gl.glOrtho(0, width, height, 0, -1, 1);
        }

        /**
         * Called by the JOGL rendering process if/when the display mode
         * is changed.
         *
         * @param drawable The GL context which has changed
         * @param modeChanged True if the display mode has changed
         * @param deviceChanged True if the device in use has changed
         */
        public void displayChanged(GLDrawable drawable,
                                   boolean modeChanged,
                                   boolean deviceChanged) {
                // we're not going to do anything here,
                // we could react to the display
                // mode changing but for the tutorial there's not much point.
        }

}
```

As you can see, using GL from inside the initialise() and display() methods is fairly simple. The most complicated part of the game window is the initialisation of the main window, this all takes place in startRendering(). First off lets look at how we initialise JOGL.

```
canvas = GLDrawableFactory.getFactory().createGLCanvas(new GLCapabilities());
canvas.addGLEventListener(this);
canvas.setNoAutoRedrawMode(true);
canvas.setFocusable(true);
```

We first create a canvas based on a set of GLCapabilities. These capabilities can define exactly what abilities the OpenGL context we create must have. For our purposes we'll just use the default set which suits us fine for some simple 2D graphics. Next, we set ourselfs up as the GLEventListener so we'll get notified when intiailisation and rendering happen. Finally, we configure the canvas to be the focused component and to require active redrawing.

Just like in the Java2DGameWindow we create a Frame and add the created canvas to it. The last thing we need to do is start some sort of game loop. Since we want our GLCanvas to be actively redrawn we can use a utility class provided from JOGL, using these two lines:

```
Animator animator = new Animator(canvas);
...
animator.start();
```

The animator will cause the GLCanvas to be initialised and to be actively redrawn as fast as possible, which is essentially what we do in Java2DGameWindow except we don't have a useful utility class to do it for us.

And thats it! Setting up an OpenGL rendering surface is very simple using JOGL. However, now we need to look at sprites.

JOGL Sprites

To use images in OpenGL they must first become textures. What we're going to do is create a texture for each image we want to use in the game. Then we'll draw a polygon (a quad to be exact) on which we'll put our sprite image.

Loading Textures

Loading textures isn't a simple matter. First you need to load the image from disk, we're lucky in Java, the ImageIO class can do this for us. Next we need to create a block of memory to store the image in a format that OpenGL can

understand. Finally, we need to tell OpenGL where the memory is and to load it as a texture. The details of this procedure are outside of the scope of this tutorial. The source code includes a class that will allow us to load textures (which has been based on a class provided in the **Java Gaming Wiki**). The class is fairly clear and should be understandable with a bit of looking around.

Drawing the Sprites

Our final step is to create a Sprite implementation that will draw the textures we load into OpenGL. Take a look a JoglSprite, looks familiar right? Its very similar to Java2DSprite. The only ignigicant changes are how we get the image and how we draw it. Lets look first at getting hold of the image:

```
public JoglSprite(JoglGameWindow window,String ref) {
        try {
                this.window = window;
                texture = window.getTextureLoader().getTexture(ref);

                ...
```

The texture loader caches image references and so we don't have to worry about it loading the same image more than once. The texture object returned stores everything we need to use the texture.

The final change is to actually draw the sprite to the screen using OpenGL, that looks like this:

```
public void draw(int x, int y) {
        // get hold of the GL content from the window in which we're drawning
        GL gl = window.getGL();

        // store the current model matrix
        gl.glPushMatrix();

        // bind to the appropriate texture for this sprite
        texture.bind(gl);
        // translate to the right location and prepare to draw
        gl.glTranslatef(x, y, 0);
        gl.glColor3f(1,1,1);

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

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

First we retrieve access to the GL context from the window we're rendering into. Just like in the Java2D version we do this inside the class to prevent the caller having to worry about where the sprite is being rendered.

Next we indicate to OpenGL which texture we're using for this sprite. The texture object returned from the texture loader has a simple bind() method to allow us to do this. Once we've told GL which texture to use we simply draw a quad where we want the sprite to appear. For each vertex on the quad we specify a texture coordinate, normally called texture mapping. Note, the use of texture.getWidth() and texture.getHeight(), this is to adapt the texture mapping for the size of texture that has been allocated. When textures are created they must be certain sizes and only a proprotion of them may need to be mapped across the quad. The texture object returned from the texture contains information decribing what proporption of the texture has been used. This is used to map the texture exactly across the quad.

So, to draw a sprite in OpenGL we load a texture, draw a quad and map the texture across it. There are many other ways of drawing similar sprites in OpenGL (glBitmap for instance). There are also ways to optimize these calls using GL lists and vertex buffers. However, for simple 2D games most this isn't really required.

Finishing Off

As you can see, when completing the game I've added a small window to allow you to choose your rendering type. This code for this is very simple and can be found in the main() method in **Game**
Hopefully this tutorial has been of some help. Personally, I've only used JOGL, hence the tutorial being oriented towards it, however I've heard LWJGL is a great API and I'd really recommend checking it out.

Another interesting point is that in Java 1.5 the Java 2D is going to support OpenGL, that is it will be possible to render the Java2D graphics using OpenGL. Hopefully this is going to make using OpenGL for 2D graphics in Java redundant.

If you have any comments or corrections feel free to mail me **here**

Exercises for the Reader

Port to LWJGL

It should be fairly simply to provide a rendering layer that supports LWJGL instead of JOGL. Implementing a new rendering layer should help to understand the abstraction and from what I hear LWJGL is well worth understanding.

Credits

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

---

previous **up** **next**

Space Invaders 102 - Timing and Animation in Java Space Invaders 104 - Rendering in LWJGL

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

Awesome tutorial, one of the

Awesome tutorial, one of the best I found since ages.
Just one more question, is there any chance there will be an update to get it compiled with the newest build of jogl and its new package structure?

Can't get it myself because some methods seems to be changed...

By Anonymous (not verified) at Wed, 2006-11-22 19:58 | **reply**

I'd suggest using the source

I'd suggest using the source from SpaceInvaders 104 and LWJGL. I'm not using JOGL in my everday stuff any more.

Kev

By kevin at Thu, 2006-11-23 01:17 | **reply**

Thank you, excellent

Thank you, excellent tutorials - not just this one. I've been looking for stuff like this in ages and these just hit the spot :)

By Anonymous (not verified) at Fri, 2008-10-24 18:37 | **reply**

in TextureLoader.java - I

in TextureLoader.java - I added the line (164)
System.out.println(textureBuffer.toString());

to print the byte buffer :
java.nio.DirectByteBuffer[pos=131072 lim=131072 cap=131072]

Exception in thread "AWT-EventQueue-0" java.lang.IndexOutOfBoundsException: Required 131072 remaining bytes in buffer, only had 0
at com.sun.gluegen.runtime.BufferFactory.rangeCheckBytes(BufferFactory.java:274)
at com.sun.opengl.impl.GLImpl.glTexImage2D(GLImpl.java:21146)
at javax.media.opengl.DebugGL.glTexImage2D(DebugGL.java:9033)
at org.newdawn.spaceinvaders.jogl.TextureLoader.getTexture(TextureLoader.java:174)

I'm using version 1.1.1a jogl.

Any Idea?

I added the println just to see the buffer before calling glTextImage2D.

Kev: Looks like the buffer needs to be flip()ed

By Ken (not verified) at Thu, 2010-06-03 16:03 | **reply**