Practical 1 – Graphics & Java

This practical is concerned with looking at aspects of displaying graphics using Java. We will develop some simple applications that display and move images about the screen.

Before we start, download the file, Code.zip, from the Canvas Practicals page, to a suitable folder on your H drive (it will be assumed that you have called this directory N6 in the following sheet), and uncompress it there. Inside this you will find the 2DCode directory, which in turn contains a number of subfolders containing Java code files. We will use the contents of the 2DCode directory as the starting point for the 2D practicals that follow so please refer back to this directory in subsequent practicals.

Simple Java Graphics

1. We will start by looking at the simple 'Hello World' Java console application shown in lectures. Create a project using using your preferred IDE (e.g. BlueJ or Eclipse) and add all the Java files found in the SimpleScreen directory to your project. (If you are using BlueJ, you will need to make the 'FullScreenTest.java' file your 'Main' file).

   Within your IDE, look at the contents of the 'FullScreenTest.java' file and note how the format fits the layout of the console application structure that was discussed in the lectures. Near the top of the file we have the method `main` which is where we create an instance of a `FullScreenTest` object called `test`, pass it a display mode to use and tell it to start running by calling the `go` method. The method `go` sets up the screen with a blue background, white foreground and a default font. It then creates a `SimpleScreenManager` object which it uses to initialise the display. Having got tired of all this work, it tries to go to sleep for 5 seconds and then quits.

   Since `FullScreenTest` is actually an extension of a 'JFrame' object (see the top of the file), it will be asked by the Java Virtual Machine to repaint itself occasionally. This occurs using the Model/View/Controller architecture you should already have learnt in previous modules. The JVM will have a reference to this 'JFrame' based object and will call its `paint` method when it thinks it needs displaying, so although we do not explicitly call `paint`, we can usually rely on the JVM to do it for us (hopefully before the application quits).

   The upshot of all this is that we put all our program display code in the `paint` method and stand well back. Compile this project and run it. You should get a blue screen with some white text and an oval.

2. Now that you have this application working, take some time to play with the code in the `go` and `paint` methods of `FullScreenTest`. Try changing the background and foreground colours, refer to the lecture notes to draw and fill different shapes on screen (try all the different methods suggested in the lectures on 'Graphics & Java'). You could also copy the code regarding anti-aliasing and apply it to your text or graphics.

3. The next step is to add images to our display since drawing objects using graphics primitives is quite hard work and it would be quite useful if we could do some of this work offline using a good graphics package. A separate set of Java classes has been provided for you to examine the process of drawing images.

   The relevant files are in the directory DisplayImages. Close your original project and create a new IDE project using the code DisplayImages. This project uses the same `SimpleScreenManager` class as before and the `FullScreenTestImage` class contains the same general structure as `FullScreenTest`.

   The main difference in this project is the declaration and initialisation of two `Image` objects. The `Image` objects are declared at the top of the `FullScreenTestImage` class file, just below the initial class declaration. The actual image contents are then loaded

when we call the `go` method. We then have to wait until the `paint` method is called before they are displayed.

In principle we could load and display the images every time the `paint` method was called but this would be very inefficient. The approach taken here is to declare them as attributes of the class `FullScreenTestImage`, load them when we start up and then refer to them when we need them.

Try displaying more copies of the image around the screen. You could also try loading and displaying other images. The images used in this application are all located within the 'images' sub-folder. If you have any other images you would like to use, put them in here and make sure you use the same references as the other image files, for example:

```
Image myImage = new ImageIcon("images/myImage.png").getImage();
```

Animation

Now that we have managed to draw images and graphics, we will look at animating our picture. The code for this section of the practical can be found in the directory Animation. As before, create a suitable IDE project for this code then set AnimationTest1.java' file to be the 'main' file.

This code relates to the animation example that was demonstrated in the lecture and consists of three classes. The first class, `SimpleScreenManager` is the same class you have been using in the sections above. We have a new class called `Animation` that manages the separate animation frames and decides which animation frame should be shown at a given point. This class also contains a private class definition for the class `AnimFrame` which it uses to handle frame information. This is the code that was discussed in the previous lecture on Animations. The final class in this example is the `AnimationTest1` class that is our controller and contains the code that starts the whole application going.

Open 'AnimationTest1.java' and look at the initialisation code in the `main` method. Confirm that this fits the format for the console application code that you saw in the lectures. Below the declaration of the `main` method, you should see the declaration of 3 attributes for the `AnimationTest1` class and a declaration of a constant called 'DEMO_TIME'. The three attributes are used to hold state information for the class, in particular, they provide a screen manager to use, a background image to paint and an `Animation` object to store all the animations in.

Below these attributes, you should find the `loadImages` method which is responsible for loading in each of the images we are going to use. Once this method has loaded in the raw images, it inserts references to them into the `Animation` object `anim` with each image being allocated a specific time (in milliseconds) for a given frame to be shown. Note that we are only storing references to images in the animation object and not copies of the images themselves – this is a more efficient process than using copies and allows us to re-use the same image many times with little overhead. The other key methods in this class are `run`, `animationLoop` and `draw`. Starting with the `run` method, examine how these methods work and inter-operate.

Compile and execute this program and check that it runs the way that you expect it to. Now go back to the method `loadImages` and try re-arranging the order in which the animation frames are added to the `Animation` object. See if you can observe your changes.

Once you have looked at adding or re-ordering the animation frames, look at the code in the 'Animation.java' file and relate this back to the material that was discussed in the lectures.

You have probably noticed that the rendered image flickers quite a bit. This is due to the continual drawing of the background and foreground images on the screen. At times, you see

the foreground and at other times you see the background. The solution to this problem is to use the double buffering approach as discussed in the lecture. Double buffering draws the complete image in a buffer, then copies the buffer to the screen. This requires us to use a more advanced version of the screen manager class that automatically tries to enable double buffering.

Close your current project and create a new project folder called 'AnimationFix' using the contents from the directory AnimationFix for the source of this new project.

The file Animation.java is the same as in the previous project however we are now using a new screen manager class called `ScreenManager`. The new screen manager has a number of useful methods that enable an application to investigate suitable screen modes and refresh rates. The important addition for our purposes is the creation of a `getGraphics` method which returns a reference to a double buffered `Graphics2D` object (Graphics2D is an extension to the default `Graphics` class that we examined earlier). Now when we want to display graphics, we ask for a reference to the `Graphics2D` object, render our picture and then call `update` to flush (send) this image to the screen. If you open the file 'AnimationTest2' and inspect the `animationLoop` method, you can see this process in action. Look at this code, then compile the project (If you are using BlueJ, you will need to set the file AnimationTest2.java as the main file). If all goes well, the flicker should have disappeared.

Depending upon the capabilities of the graphics card and configuration of Java, the above double buffering approach does not always work. You should notice that there is a further Java file available called AnimationTest3.java. This uses the alternative method of directly implementing double buffering that was discussed in the lectures.

Build a new project consisting of Animation.java, ScreenManager.java and AnimationTest3.java, then look at the animation loop code in this file and make sure you understand what it is doing. Now try copying the alterations that you made in the `loadImages` method of the previous flickering animation code to this project and see if your chosen sequence is more obvious.

Animations are often supplied as 'sheets' with all the animation frames in them. To save you time and effort, a couple of methods have been added to the Animation class to load a set of images from a sheet of animations. The method to call from the loadImages method of AnimationTest is *loadAnimationFromSheet*. If you have an animation object called *anim*, a sprite sheet called *spritesheet.png* (that you have put in your images directory and which has the individual images arranged in a grid with 5 columns and 4 rows) and a frame duration of 100 milli-seconds, you would use it as follows:

```
anim.loadAnimationFromSheet("images/spritesheet.png",5,4,100)
```

There is a small sprite sheet called landbird.png that you should find in the *images* subdirectory of AnimationFix that you can use to test this method. The individual images are arranged in a grid of 4 columns and 1 row. To finish off, see if you can locate potential animations for your assignment and try to get one of them to animate correctly either by adding single animation frames or using a sprite sheet.