Practical 3 – GameCore & Transforms

At the start of this practical we are going to look at the GameCore class in some detail and then move on to examine the use of transforms. In the next practical we will look at collision detection and tile maps. Combined with the sounds and event handling, you will then have looked at all the elements necessary to write your own 2D game for the assignment.

Game Core

We are going to start this practical by looking at how to use the GameCore class discussed in lectures to form the basis of a game. Create a new project using the files available in 2DCode\GameCore. Compile this project and run it by using GCTester as the main file. You should see the words 'Time Expired' and a counter appear on the screen for 5 seconds and then the program will stop.

Now let us look at what is happening here. The GameCore class in the newly introduced *game2D* package is actually performing most of the operations that you looked at in your previous practicals. This *game2D* package has been created to hold all the useful classes that we have been looking at and working with in the lectures (your assignment will also use this package but with added functionality to make it easier for you to develop games).

The whole process is started when the 'run' method is called in 'main'. 'run' will call the 'init' method to see what needs to be set up, then it will start an animation loop. Within the animation loop, the methods 'update' and 'draw' will be repeatedly called. In order to do something useful, we therefore need to override 'update' and 'draw' such that our version of things takes place instead of using the empty methods defined in GameCore.

Look through the GCTester.java file and observe how the above process has been followed in order to set a counter for the total time expired to 0 in the *init* method, how the current total time is displayed in the *draw* method and how we check to see if the program should terminate in *update*. Note also that we can use the inherited method *stop* in order to quit the program.

1. Our next task is to try and re-create some of the effects we achieved with our previous practical work, except this time we are going to do it using the GameCore framework. The first thing we will look at is adding a moving Sprite to our 'game'. To do this we will need to declare an Animation attribute called 'anim' and a Sprite attribute called 'rock' to our GCTester class. Add these attributes to the GCTester class by declaring them just after the GCTester class declaration.

2. Now create an instance of the 'anim' Animation in the *init* method, just after the initialisation of *total* to 0 and an animation frame to this newly created instance containing the picture 'images/rock.png':

```
anim = new Animation();
anim.addFrame(loadImage("images/rock.png"), 250);
```

   Once you have initialised the 'rock' sprite with the above animation, try setting its initial location and motion in the *init* method using the following commands:

```
rock = new Sprite(anim);
rock.setPosition(70,70);
rock.setVelocity(0.1f,0.1f);
```

   A further addition has been made to the default Sprite class via the inclusion of the methods *show*, *hide* and *isVisible*. These methods enable you show or hide a sprite (and find out which state it is in) without having to create a new one or destroy it each time you want to change its visibility. This can be very useful when you want to re-use a sprite object at a later point but do not

want the user to see it at present.

In order to see the 'rock' sprite on the screen, we now need to draw it. Another two methods have been added to the default Sprite class that you saw in the lectures, one of which we will come to later. The method that is of most use to us at present is the new Sprite *draw* method that takes a reference to a Graphics2D object and takes care of the required draw operation. By calling this method within our GCTester *draw* method, we can get the rock drawn on the screen. Insert the following code in the GCTester *draw* method to try this out.

```
rock.draw(g);
```

Now compile your program and check to make sure that you observe a rock in the top left corner of your screen.

3. You are now probably asking yourself, 'I thought I told that rock to move, why is it stationary?'. The reason for this is that we have not given the 'rock' object an opportunity to update its position in the *update* method in GCTester. This is easily fixed by putting the following call in the 'update' method:

```
rock.update(elapsed);
```

Add this method call and then compile and run your program. Your rock should now move slowly towards the bottom right hand edge of the screen.

4. Our next step is to add keyboard event handlers so that we can change the behaviour of the rock. In the first instance, we want to stop the rock's motion when we press the 'S' key and start it again when we press the 'G' key. In order to do this, we need to alter the GCTester class so that it implements the 'KeyListener' interface.

By default, GameCore already implements the KeyListener interface in order to detect the user pressing the 'Escape' key. We therefore only have to override the particular method we are interested in (all the others are already present in game2D. In our case we only need to provide a new *keyPressed* method to GCTester so add the following code into it:

```
public void keyPressed(KeyEvent e)
{
   switch (e.getKeyCode())
   {
         case KeyEvent.VK_ESCAPE: stop(); break; // Stop game loop
         case KeyEvent.VK_S: rock.stop(); break; // Stop the rock
         default: break; // Unused key event
   }
   e.consume();
}
```

The package which contains the information for the event handler interfaces is java.awt.event so make sure you also add the following import statement to the top of GCTester.java file:

```
import java.awt.event.*;
```

Compile and run your program – check to see that you can stop the motion of the rock sprite and also check to see if pressing the Escape key stops the program. Note also that an extra method is available for sprites that will cause them to stop moving, imaginatively called *stop*.

5. Once you have confirmed that pressing the Escape key stops your game, you can comment out the line in update that stops the application after 5 seconds:

```
if (total > 5000) stop();
```

6. The next thing you should do is add an extra key check to see if the user pressed the 'G' key and if they did, start the rock sprite moving again. Look in the *init* method for an example of how to set the rock sprite's velocity parameters.

7. Instead of starting and stopping the motion of the rock, try to add a 'Pause' function that only updates the game state if the game is not paused. Hint: You will want to declare a class attribute called 'paused' at the top of GCTester and put a line at the start of the *update* method that looks something like:

```
if (paused) return;
```

   You will then need to set up a key press detection that toggles the state of the boolean variable 'paused' where 'paused' has an initial state of 'false'. To toggle this boolean variable, you can use:

```
paused = !paused;
```

8. We are now going to add an extra method that will cause our rock to come back on the opposite edge of the screen that it came off, such that it is always visible. To do this, create the following method in GCTester.java called checkScreenEdge that takes a single 'Sprite' parameter:

```
public void checkScreenEdge(Sprite s)
{
   if (s.getX() > getWidth()) { s.setX(0); }
   if (s.getY() > getHeight()){ s.setY(0); }
}
```

   This code shows how to check for the Sprite moving off the right and bottom edges of the screen. You should add further own code to test for the remaining screen edges. Once you have done this, check to see if your code works by putting a call to *checkScreenEdge'* in the update method and passing it the reference to the rock Sprite object as follows:

```
checkScreenEdge(rock);
```

   You should generally get into the habit of creating methods that are given one or more Sprites as parameters which then perform some operation on those Sprites. It will make it much easier to work on your assignment if you follow this approach since you can re-use these methods for all the different sprites you will have in your game.

Practical 3 – GameCore & Transforms

Transforms
Our next task is to investigate using transforms to draw our sprite. This will require us to stop using the *Sprite.draw* method in *GCTester.draw* since we would like a little more control over how our sprite is drawn. We are going to investigate 3 possible transforms:

- Translation (move to a particular position)
- Scaling (grow or shrink an image)
- Rotation (spin an image)

The first translation will replicate the drawing activity that we are currently doing in Sprite.draw such that you should see no obvious change in behaviour.

1. Begin by commenting out the call of *rock.draw(g)* in the *GCTester.draw* method and add the following code instead.

   ```
   AffineTransform transform = new AffineTransform();
   transform.translate(Math.round(rock.getX()),Math.round(rock.getY()));
   g.drawImage(rock.getImage(),transform,null);
   ```

   The *AffineTransform* class is part of the package *java.awt.geom* so you will also need to add an import statement for this package at the top of your 'GCTester.java' file. The relevant import statement is as follows:

   ```
   import java.awt.geom.*;
   ```

   Compile and run your program. You should observe no difference in the motion of the sprite. You are probably asking yourself 'So what was the point of that?' The reason we use transforms is that we can add them together to get interesting effects, as follows.

2. After the call to 'translate(…)' in the above code, add the line:

   ```
   transform.scale(0.5f,0.5f);
   ```

   This will shrink the image used to render the sprite by 50%. Compile and observe this change. Now try altering the above two parameters to see what effects you can create - for example you could stretch the image in the X dimension, whilst shrinking it in the Y dimension with the call scale(2.0f,0.5f).

3. We are now going to look at the very useful 'rotate' transform. This transform takes a rotation parameter in radians and rotates the image to match that angle. If you do not like using radians, you can use the Math.toRadians(x) method to convert a value 'x' in degrees to the same value in radians. Add an attribute to the GCTester class at the top of file called angle of type double:

   ```
   private double angle=45;
   ```

   In the 'init' method of GCTester.java, initialise this value to a particular angle, e.g. 90 (for 90 degrees). Now add the following code to perform the transform call after the `transform.scale` call in the *draw* method of GCTester.

   ```
   int width = rock.getImage().getWidth(null);
   int height = rock.getImage().getHeight(null);
   transform.rotate(Math.toRadians(angle),width/2,height/2);
   ```

Compile and check to see if you can observe a change in the rotation of the rock (you may want to enlarge the rock so you can see the effect more easily). Note that the rock will not spin, it will just be at a different angle to the default in the png file. We will get round to spinning it later on in this practical.

You are probably wondering what the extra 2 parameters in the rotation call are referring to. These relate to the point in the image about which the rotation will occur. By default, the rotate transform will rotate an image around its top left corner (0,0). This isn't very useful for us so we have stated that the rotation should occur at the centre of the image (i.e. half its width and half its height).

4. For our next task, we shall add some interaction to this process. It would be quite nice if we could rotate our rock to any angle we like. To do this we will need to add two more checks in our key event handling code together with appropriate responses. Add the following lines to the case statement in your *keyPressed* event handler at the point after the checks for pressing the 'S' and 'G' keys:

```
case KeyEvent.VK_RIGHT: angle = (angle + 5) % 360; break;
case KeyEvent.VK_LEFT: angle = (angle - 5) % 360; break;
```

This will increase or decrease the angle of rotation by 5 degrees. We use the modulus operator '%' in order to make sure that the angle remains in the range 0 to 360. Compile and test your code to check this behaviour.

5. If you would like an extra challenge, see if you can add a line of code to the *update* method that will change the angle based on the elapsed time since the previous update so that the rock slowly tumbles as it moves along.