**Lab 5 - Layering & Text**

**Step 1 – Input continued**

Right now, both of our paddles are controlled by the same key, but let’s change that so we can have a real 2-player experience. Our left paddle will be controlled by the **W** and **S** keys like before, while our right paddle will be controlled by the **Up** and **Down** arrow keys. To do this, we’ll need have our paddles move based off of keys we pass to them. Let’s go to our **PongPaddle** class.

The keys on our keyboard represented by ints, so our **PongPaddle** class can simply ask for the int values of the upward and downward keys we want to check. Do the following:

* Add two private int variables to our class, one for the upward and downward key
* Add two additional parameters to our constructor that are the upward and downward keys we want to use for this paddle. Set our private variables to those parameters.
* In our **update()** method, change the two **Input.keyDown()** calls to use our upward and downward private variables.

Once this is down, go back to the **PongGame** class. In order to use the **Input** class in here, we’ll need to add the following import statement:

import com.radirius.mercury.input.\*;

For our left paddle’s constructor, pass in **Input.KEY\_W** and **Input.KEY\_S**; while for our right paddle’s constructor, pass in **Input.KEY\_UP** and **Input.KEY\_DOWN**. Compile and run your program, then try moving both of the paddles separately with our new keyboard keys.

**Step 2 – Text**

Let’s add a little bit of flair to our game. We currently have no way to keep track of our score for the game, so let’s start by adding some text that can display each side’s points. Go to Edmodo and look for the “Game Development – Day 5” post. Download the zip file and extract all of the contents into your folder. We’ll be using the file “**font.ttf**” to display the text for our scores.

Let’s create a new class called **PongScore**. Add the following imports to the top of the class:

import java.io.\*;  
import com.radirius.mercury.framework.\*;  
import com.radirius.mercury.graphics.\*;  
import com.radirius.mercury.graphics.font.\*;  
import com.radirius.mercury.resource.\*;

Our **PongScore** class is going to use a special class called the **TrueTypeFont** to help display our text. The **TrueTypeFont** object stores the type of font to use when we want to draw text to the screen. You will need to do the following for our **PongScore** class:

* Create a private **TrueTypeFont** variable called **font**
* Create two private int variables, **leftScore** for the left score and **rightScore** for the right score

Once we have that, let’s create a constructor for our **PongScore** class. Use the following code:

InputStream fontStream = Loader.getResourceAsStream("font.ttf");  
this.font = TrueTypeFont.loadTrueTypeFont(fontStream, 100);

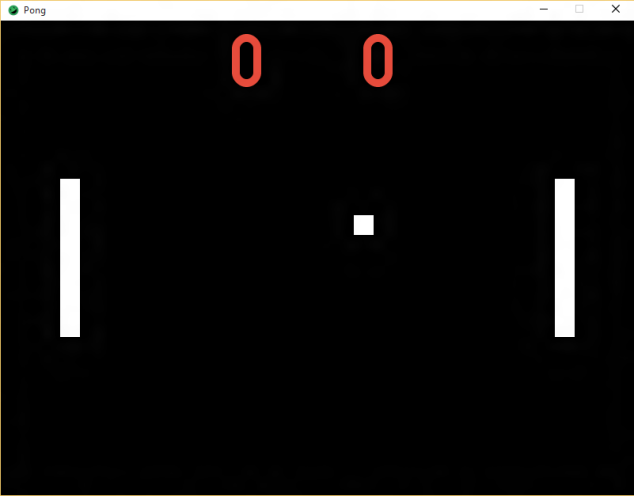
Let’s talk about what this code does. The first line is very similar to how we loaded in an image file, except we are looking the font file. The second line creates a new **TrueTypeFont** object from the file we just read in.

Let’s now add a **render(Graphics g)** method to our **PongScore** class. Use the following code inside of it:

g.setFont(this.font);  
 g.setColor(Color.RED);  
 g.drawString(this.leftScore + " " + this.rightScore, 290, 0);

The first line of code sets the font to use with our graphics renderer and the second line sets the color to be RED. The third line instructors the **Graphics** object to draw the specified string to window at the **X** and **Y** coordinates. The string that will be printed will show the left score, **5 spaces**, and then the right score at the coordinate (290, 0).

Let’s now go back to our **PongGame** class. Go through our full set of steps to add a new object to our **PongGame** class (create a private variable, create a new **PongScore** object and assign it to our variable, call the **render()** method). Compile and run your code and you should see the following:



Before we move on, let’s quickly go through how the **Graphics** class works when you try to draw something. When you call the **draw\*()** methods, such as **drawShape()**, **drawTexture()**, and **drawString()**, the **Graphics** class will draw each of those calls **in the order you specify**. Sometimes this isn’t a big deal, other times it can mean that certain things can be hidden. Let’s look at a quick example:

Let’s say I decide to draw 3 **Rectangles** that are **BLUE, WHITE,** and **BLACK** and in that order. Let’s also say that these **Rectangles** all **intersect** with each other. When they get drawn, this is what will happen:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| BLUE Rectangle drawn | WHITE Rectangle drawn | BLACK Rectangle drawn |

Notice how each time a new **Rectangle** is drawn, it overlaps **ON TOP** of the previous one? This means that the first object that is drawn will always be **AT THE BACK**, while the last object drawn will always be **AT THE FRONT**. There is one caveat, this sort of “stacking” of drawn objects only occurs if the objects intersect with each other. If there is no intersection, then they’ll be drawn normally.

So we have numbers, great! But they don’t change…Currently, if the ball goes off to the left or right end, the ball just keeps going and we never see it again. Instead, if the ball goes off to either end, we should reset the ball in the middle and update the score for the appropriate side. Let’s go back to our **PongBall** class and add a new method called **checkPointCollision( int width )** which returns an int. The return value of the method is determined this way:

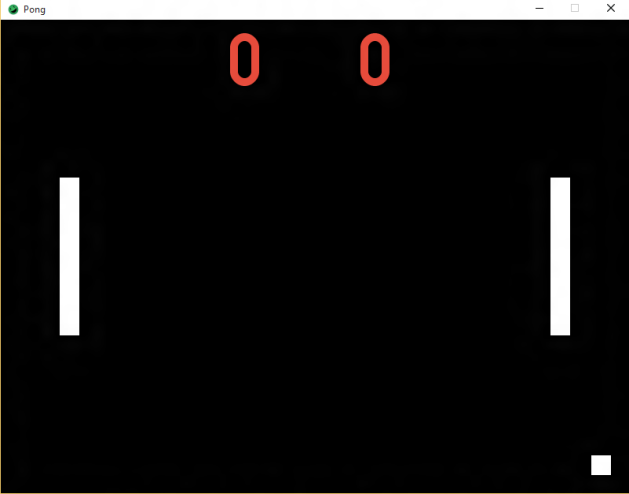
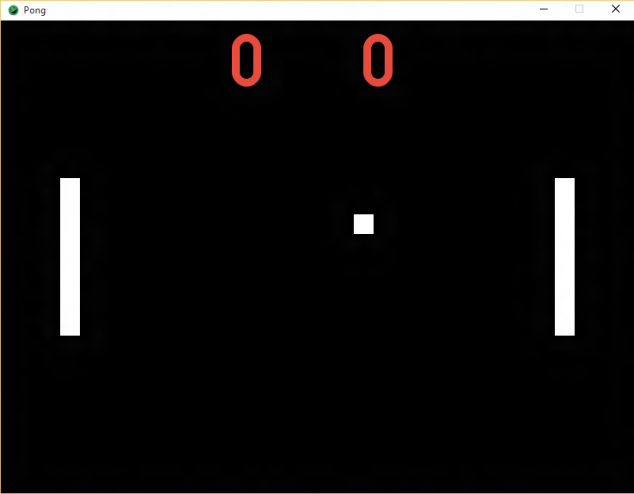
* If the ball has touched the left wall, return 1 (the right player got a point)
* If the ball has touched the right wall, return -1 (the left player got a point)
* If neither of the above is true, return 0 (no point)

In order to check the above, we can use a similar setup to how we did in the **checkWindowCollision()** method. Just like there is a **getY()** and **getY2()** method, there is also a **getX()** and **getX2()** method. Think about what we would need to change to make it to work for this scenario.

Also, if the ball has touched **either** wall, we want to reset the ball back to the center of the screen. We so far know the **translate( int x, int y )** method, which **MOVES** the **Rectangle** by the amount of **X** and **Y**. There is also another method called **translateTo( int x, int y )**, which **MOVES the Rectangle TO** the positon (x, y). In order to move the **Rectangle** back to its starting position, we’ll need to go to the constructor and **save the values** for the parameters **x** and **y** off into separate values so we can use them later. To quickly recap, we need to do the following:

* Create two private int variables to save the initial position of the **PongBall** in the constructor
* Create the **checkPointCollision( int width )** method
  + If the ball touched either wall, move the rectangle back to the starting position
  + If the ball has touched the left wall, return 1 (the right player got a point)
  + If the ball has touched the right wall, return -1 (the left player got a point)
  + If neither of the above is true, return 0 (no point)

Once this is done, go back the **PongGame** class and the **update()** method. Call the **checkPointCollision()** method we just made and save the return value into a variable (we’ll use this in a little bit). Compile and run your code. Once the ball goes off to either side, you should see the ball pop back up in the center:

Last thing we need to do now is update the score we display. Go to our **PongScore** class and add a method called **updateScore( int point )**. For this method, if **point** is -1, we’ll add 1 to the **leftScore**; where as if **point** is 1, we’ll add 1 to the **rightScore**. Once this is done, go back to **PongGame**. Remember the int return value we saved off from **checkPointCollision()**? Call the **updateScore( int point )** method and pass that int in as a parameter. Compile and run your code and you’ll see the points update now:

