**Decision Day – 19/02/22 – Computing Science, Software Engineering, Games Development**

**Setting Up**

Log in to a machine. (Username:ysjguestX, password:udkhirdy)

Open this link in a browser, you should see a Firefox icon on the bar at the bottom of the screen.

<https://git.ysjcs.net:8888/a.guest/decisionday270120/blob/master/processing.pde>

Down load the file by clicking on the download button. The download button looks like a cloud with a down arrow on it.

**Processing**

Open Processing – icon looks like 13 in white on a black background.

Open the File menu, Select Open.., Select processing.pde from the file list and press Open. Click yes when you get a pop up window asking about a folder.

Press the Play button in Processing to run the application.

**The Game**

The application is a simple shooting game. Move the mouse over a moving target and left click to shoot it. When you’ve shot a few targets you can close the game window by pressing Escape.

**The Processing Code**

Have a look at the code in the Processing window.

The main structure of the code has two parts

setup()

draw()

The instructions in setup() are carried out once when the game first starts. All setup() does in this game is set the size of the game’s window.

The instructions in draw() are carried out over and over until the game exits. Each instruction in draw() is carried out in turn and then when they are all done it goes back to the start of draw and does it all again.

I’ve split draw() into three parts – input(), update() and render(). This is how most games are structured. This makes it easier to see what is happening and keep track of what happens when.

Almost every game has a **game loop**. This game loop contains instructions which are carried out repeatedly while the game runs.

Typically the first thing that happens in the **game loop** is the code checks for any key presses, mouse movements or controller inputs. In this game the **input()** function does this. It checks to see if the mouse button has been pressed and if it has checks to see if it is on target.

Next in the **game loop** is the **update()** function. This is where games update the state of the game based on what has happened. So if a button to move left has been pressed then the data describing the position on the player is updated to show its new position. Here we also carry out checks to make sure any movement is valid, check for collisions, check on timers and lifespans, etc. Complex games usually break the update down in to steps – effects of input, followed by effects of physics/time and then AI instructions of enemies for example.

Finally the **game loop** draws the game to the screen in the **render()** function. This simply goes through everything that should be visible and draws it.

**Exercise 1**

1. The line “size = 50;” (on line 88) determines the size of the target. Try changing the 50 to 75 and running the code. The target should be much bigger.
2. Now try changing the 75 to 25. The target should be much smaller.
3. *size* is a variable, we use it to store the size of the target. We use this size not only to draw the target but also determine if we are clicking in the right place to shoot it.
4. We can make the size random if we like. Change line 88 to size = int(random(50)); and run the code.
5. The target should be a different size each time now.
6. Unfortunately the target is often too small. This is because the random size is anywhere from 0 to 50 and anything smaller than about 20 is too small.
7. We can fix this by changing the line to size = int(random(40))+20; Now the size will be 20 plus a random number between 0 and 40 – the size will be in the range 20 to 60.
8. Play around with the numbers to see the effects.

**Exercise 2**

1. The explosion when you hit a target is currently shades of red. If you scroll down to the bottom of the code in Processing, to the part that starts “class ExplosionPart” you will see the code that runs all the little parts of the explosions
2. Find the line that reads “red\_colour = int(random(200))+50;”
3. Colours in code are typically described with three numbers, a red, green and blue value. Each number represents an amount of that colour that goes to make up the final colour. Each number can be any value from 0 to 255 where 0 means none of that colour and 255 means as much as possible.
4. To get a red colour on screen we set the red value to the shade of red we want and set the green and blue values to zero.
5. If we want a purple colour then we need a mix of red and blue but no green.
6. If we set all three values to zero we get black.
7. If we set all three values to 255 we get white.
8. If we set all three values the same we get a shade of grey that is darker if the numbers are small and whiter if they are big.
9. Under the “red\_colour = int(random(200))+50;” is a line that reads “green\_colour = 0;”. Change the green line to “green\_colour = int(random(200))+50;” and run the game.
10. The explosion should now have changed colour.
11. int(random(200))+50 creates a random number between 50 and 250. (random(200) creates a number between 0 and 200 and we add 50 to it). Don’t worry about the int() part, that just makes sure the number is valid for a colour.
12. You can remove the “int(random(200))+50” part and replace it with a number and each part of the explosion will have the same colour (for that part of R,G or B anyway).
13. If you want your explosion to be shades of orange try leaving red set to “red\_colour = int(random(200))+50;” and changing green to “green\_colour = int(random(100))+50;” and blue to “blue\_colour = int(random(100))+50;”