FPP Report – Mark Strathie, 2147105

What? Gunslinger Game

My program is a western game. You play as a gunslinger who has to shoot an opponent called Loco. He moves around and will shoot you if you run out of ammo in your gun. You win when you kill Loco and lose if he kills you.

Brief user instructions

Simply extract the zip file. Make sure all the files (gunslinger.py, klaus.gif, background.gif, klaus1.gif, background1.gif) are in the same directory and run gunslinger.py. The python shell will show you instructions and controls. The game is played in the turtle window. It's best to maximise the screen. Here's a copy of the instructions from the game:

Instructions:
Shoot Loco 19 times to kill him and win the game. But be careful!
He will move around to avoid your shots.
It's best to maximise the turtle window so you can find him.
You have 6 bullets in your gun. Make sure you keep it loaded.
If you run out of ammo you'll give Loco time to shoot back!
He is a deadly gunslinger at can kill you in 2 shots!! Good luck.

Controls:
Use your mouse cursor to aim and left click to shoot.
Press 's' to show your score, health and ammo here in the python shell.
Press 'r' to reload 6 bullets in your gun.
Press 'Esc' to quit.

How it works

First I need to run the import function to import the turtle and random libraries, as I need this additional functionality for my game. Next I set up my counters and Boolean variable. I create the variable 'win' to make it easier to write turtle commands. I then use the bgpic command to add an image to the turtle background, "background.gif." I run the penup command to stop the turtle leaving lines behind and set it to a position to begin. I then register 2 images for the turtle shape, and set it to the first one, "klaus.gif." I then define 4 functions. 3 are called by input from the user, and 1 is called within another function. I then execute a print statement to show the user instruction in the python shell. I then set up my callback functions. They will be called when the user clicks or presses a certain key on the keyboard. I then tell the turtle to listen for these instructions with turtle.listen() and turtle.mainloop().

Data structures

I have 3 integer variables that act as counters, and 1 Boolean variable.

The ‘score’ starts from 0. It increases by 1 every time the enemy is hit. When it reaches 20 the game is won and the Boolean variable changes to False to show the game is not running anymore.

‘health’ starts at 2 and decreases by 1 every time the player fires with no ammo. The Boolean value changes to False when the health is 0, to show the game has been lost.

‘ammo’ starts at 6 and goes down by one every time the player clicks on the screen, modelling a bullet being fired. Pressing ‘r’ will reload the gun, which sets the ammo value back to 6.

Interesting / tricky / subtle aspects of the code meriting further explanation

I defined 4 functions. ‘clickResponse’ uses the global values for score, health, ammo and gameRunning. This is so the changes to these variables are kept when the heap is closed. It then starts with an if statement to check if the game is still running. If not, then the function finishes. If yes, then there’s an if statement to check if they player has ammo. If not then there’s an else statement which takes -1 from the health variable and changes the images to signal you have been shot. There’s an if statement to check if health is 0, then the turtle writes GAME OVER in dark red and the game stops running. If the player does have ammo then the ammo decreases by -1 I use an if statement to check if the cursor is over the enemy shape. If not then nothing happens. If so then the score goes up +1. I then have an if statement to check if the game is won. When score = 20 then the turtle writes YOU WIN in dark green and sets gameRunning to False. The turtle is then hidden, and I call the moveEnemy function.

moveEnemy first uses in if statement to check the game is still running. If so then it uses random to move the turtle to a random position along the x axis between 0 and 650. It then shows the turtle.

The stats and reload functions are called when the player presses ‘s’ or ‘r’. Stats simply executes a print statement showing 3 variables – score, health and ammo. Reload checks if the game is still running with an if statement. It updates the global ammo variable to 6. It then changes the images to the normal versions, which will reset them from the red images if they were changed to that within the clickResponse function.

I have 4 different images in the game. 2 backgrounds and 2 turtle shapes. A normal background and shape are used from the start of the game. If the player is hit then both images are changed to a red version to simulate blood. They change back to normal once – if the player reloads and recovers.

Particular challenges I overcame

- The player could still play the game after they won or lost. So, I decided to create a Boolean value. Most controls will now only work when gameRunning = True. When the game is finished this value changes to False.
- I had a lot of print statements so decided to use “\n” character to separate lines and combine the statements into one. Sometimes the turtle text went off the screen, so I used “\n” here as well.

- It took me some time to think of a way to get the enemy to shoot back. I eventually decided to tie it to the ammo/reload counter/function. It seems realistic to have the enemy fire when you go to fire but have no ammunition. So, I put this section of code under the section “#if player shoots with no ammo.”
- I played the game a few times to work out the difficulty level. Eventually deciding on health = 2, to give the player 1 chance to make a mistake, and score = 20 to win the game, which is 3 reloads.
- Figuring out the ‘hit’ area took some trial and error. It was easiest for me to make the turtle the target. So, I tied the ‘hit’ area to the turtle’s location. Then all I had to do was enlarge the area to the size of the turtle’s ‘shape.’