**COMP37111 Mini Project – Doggo Heaven**

**Student ID: 10333274**

# Introduction

I attempted to create a user-controlled dog model which roams freely in a plain where tennis balls drop from the sky. The project was written in the Python programming language and made use of the PyGame library. All sprites besides the tennis ball, which was acquired from Google Images, and the icon, which was acquired from the online MMORPG League of Legends, were created by me using GIMP.

There are two particles in this scene: the dog and the tennis ball. Each belonging to its particle group for easier management. The user can manipulate aspects of the game:

* the movement of the dog model
* the amount of tennis balls currently rendered
* the lifetime each tennis ball
* the ‘gravity’ physics of the scene
* toggle hitboxes of all particles

Machine specifications:

* CPU: AMD Ryzen 7 5700U (8 core / 16 thread @ 1.80 GHz base speed)
* GPU: Radeon integrated graphics
* RAM: 1 x 8GB – 3200MHz

# Limitations

The game seems to make use of multi-threaded processing to render and handle the logic of all the particles as close to real time as possible but seems to be heavy on the GPU. Figure 1 shows the game with around 60 tennis balls spawned, and the internal clock configured to render at 60 frames per second.

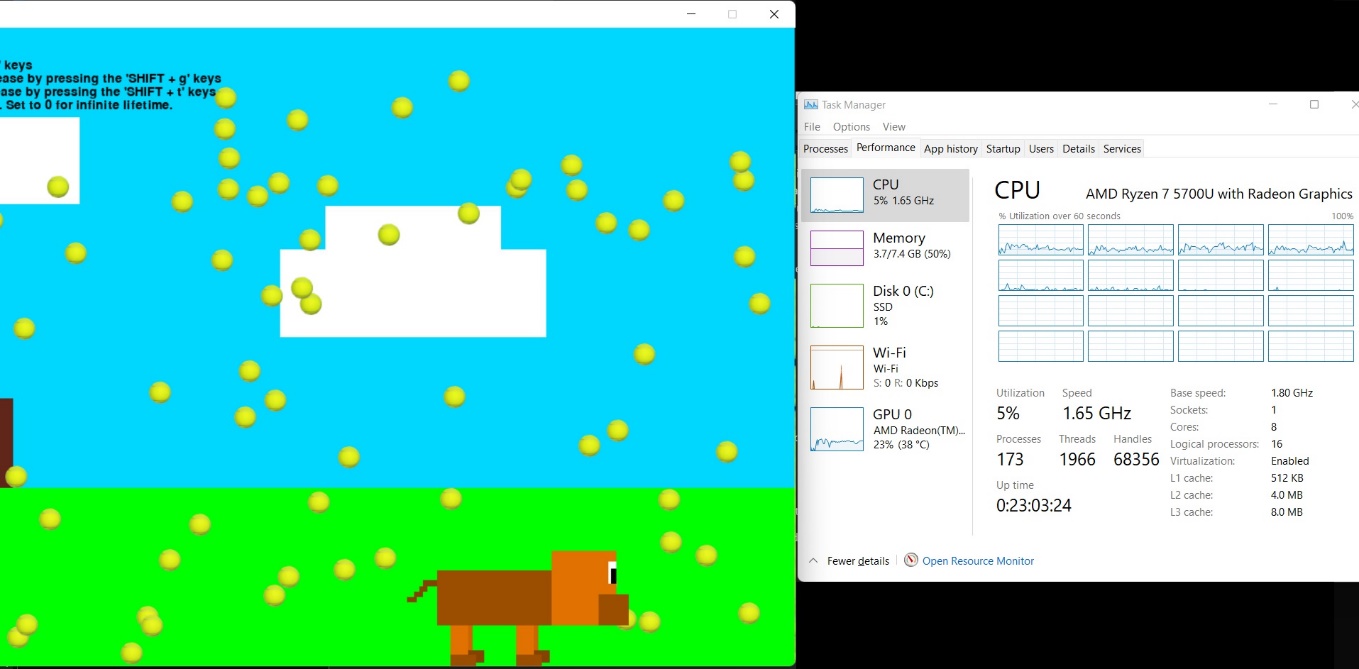


Figure 1 - Task manager shows the program makes use of multiple cores

The GPU utilisation was addressed by adding a hotkey to change the FPS of the game (more accurately the internal clock of PyGame), to increase or decrease the ticks per second. Consequently, the CPU utilisation also fluctuate in the same manner as the GPU (higher % utilisation for higher FPS, lower for less) as shown by figures 2 and 3.



Figure 2 - High graphic settings

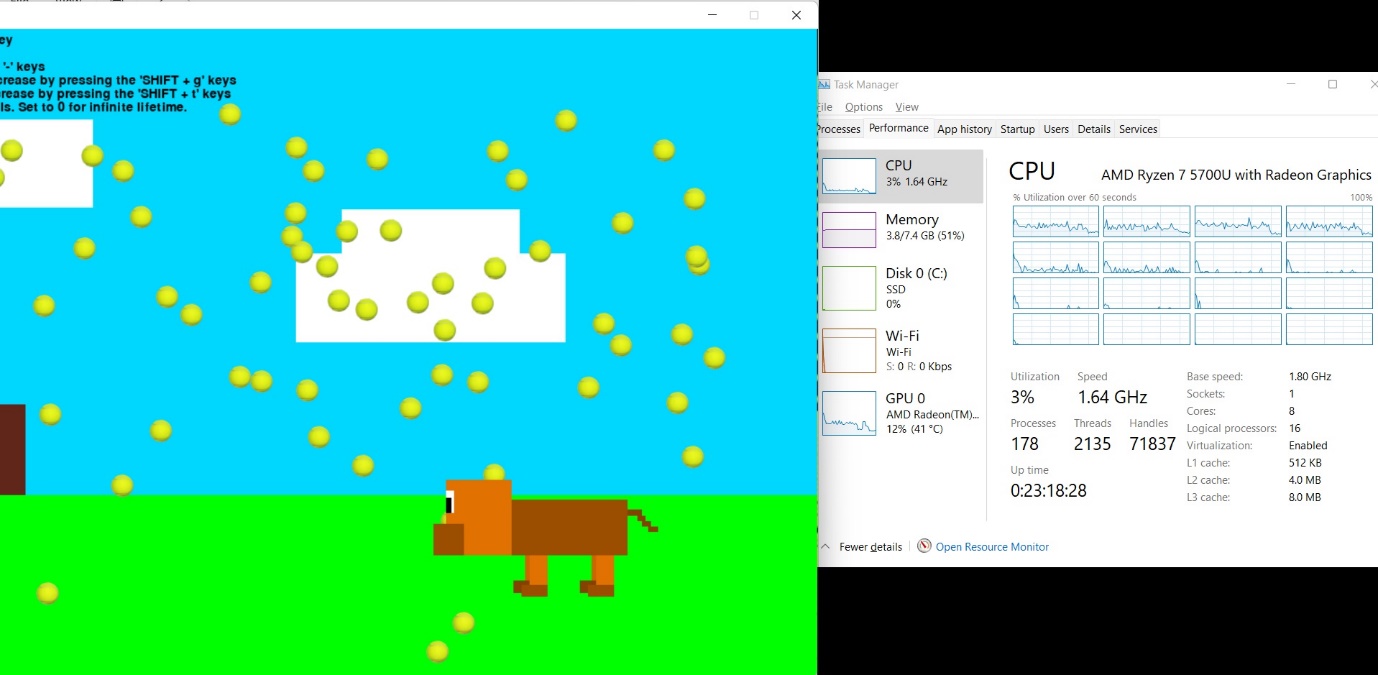


Figure 3 - Low graphic settings

It is worth noting that the CPU utilisation increases based on the number of tennis balls being rendered up until a threshold of around 120 tennis balls, in which it remains constant (with a +- 5% fluctuation). The GPU hits a plateau almost instantaneously when the FPS setting is changed (the number of tennis balls being rendered does not increase/decrease GPU utilisation). The systems main memory seems to be negligibly affected by the addition/removal of tennis ball and is therefore ignored.

I must conclude that the processes limitation might come from the game engine (PyGame) or even Python not utilising as many resources as it has available (e.g., 4 threads out of 16). The hypothesis comes from the drop of frames with no apparent increase in CPU, GPU, or memory utilisation to handle the increased number of concurrent events.

# Memory management

In the beginning I noticed that whenever I hit the reset hotkey, to reinitialise the game, the system’s main memory allocated to this process kept increasing, which gave me indications of a memory leak. After further examination of my code, I noticed that whenever I reset the game the *main* function was recreated, thus all the sprites, particle groups and particles themselves were being loaded in memory as new data structures.

To combat this, I made use of PyGame’s particle groups as class-wide structures which are written to and read from by the *main* process. This would allow me to clear these structures in between resets (refer to figure 4), and the memory seemed to continue increasing, but at a lower rate.

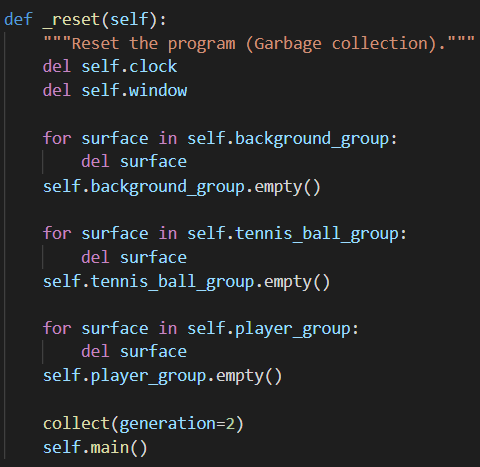


Figure 4 - Reset logic

Through further inspection, I found that the major leak in memory were the sprites that were being reloaded into memory. Hence, upon the reset hotkey press, I delete all local references of the sprites (refer to figure 5). After which the main memory remains almost constant after each reset, any further memory leaks proved to be through various local flags that were used for the player’s movement logic (e.g., jump, drop) that would cause more trouble to free, than they would consume in system resources.

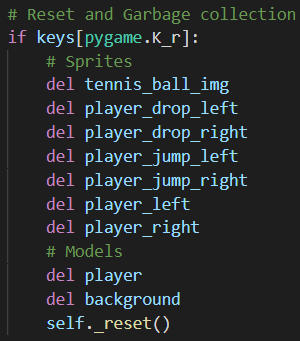


Figure 5 - Reset logic

Finally, I imported Python’s *gc* module which took care of any other resources.

# Physics

The main physical aspect of this game is the gravity logic. This is achieved via the addition of vectors. The current angle and velocity of the object (i.e., tennis ball) is updated every game tick by adding a vector of angle pi and magnitude GRAVITY defined in the games *globals.py* library. This allows me to simulate the gravitational pull of the Earth, represented as an arbitrary index.

In addition, ‘bounce’ logic was added to the tennis ball model, that detects a collision with the screen edges, and a ‘drag’ logic which updates the balls speed every tick against gravity.

Combining these allowed me to simulate a bouncing ball that reaches the bottom of the screen and bounces back up to a peak, less than its original height, eventually coming to a stop.

# Particle Collision

Particle collision is handled natively via PyGame’s API. It provides a simple data structure called a *SurfaceGroup* where each surface (or particle) is assigned to a group, and can be configured to listen to any overlaps between sprites of groups X and Y. It would be worthy to note that the groups X and Y need not be different.

After a collision is detected, a trimmed version of the bounce logic can be applied to simulate an explosion between particles, resulting in the two particles accelerating away from each other for tennis balls, or towards the direction it was pushed towards for tennis ball and player interactions.

# Improvements

Although the game is in a playable state, there are many improvements that can be made to it.

* Implementing PyOpenGL along with PyGame to create a 3D render of the scene
* Improve upon the world physics such as
  + the kinetic energy of the tennis balls (more realistic movement, loss of speed)
  + the tennis ball bounce /collision logic (make angles more accurate, consider which part of the dog it has hit and bounce accordingly instead of simply adding vectors)