COMP280 – Optimizing and Analyzing Custom Built Pac-Man Ghost AI

Isaac Wright

January 2nd, 2021

# Introduction

COMP280’s AI task asks us to create AI for the four ghosts in a version of Pac-Man, built in either Unity or Unreal Engine. Choosing to use the Unity Engine, I needed to write an AI script that would work with the behaviour tree format, having it communicate with the rest of the tree as well as providing a decent Pac-Man experience – I needed to make these ghosts not entirely unbeatable, but use a system that is difficult to avoid without causing frustration.

My script is designed to work in tandem with the other behaviour tree scripts; it derives its logic from the BTNode script given in this task. This allows the ‘evaluate’ function to be used in tandem with the behaviour tree’s blackboard, which receives and sets the target of the desired AI; each AI uses a NavMesh component to find the quickest route to their target.

Using the evaluate function additionally allows the SUCCESS or FAILURE response to be sent to the BtNode script, which will re-run or stop the script respectively, based on the script’s output.

Creating my script, I thought that the ghosts would need something to target (using the blackboard’s target GameObject) that isn’t the player. If the ghosts took the quickest path to the player at all times, the game would feel unfair, so I decided to create 25 different waypoints that the ghosts can use.

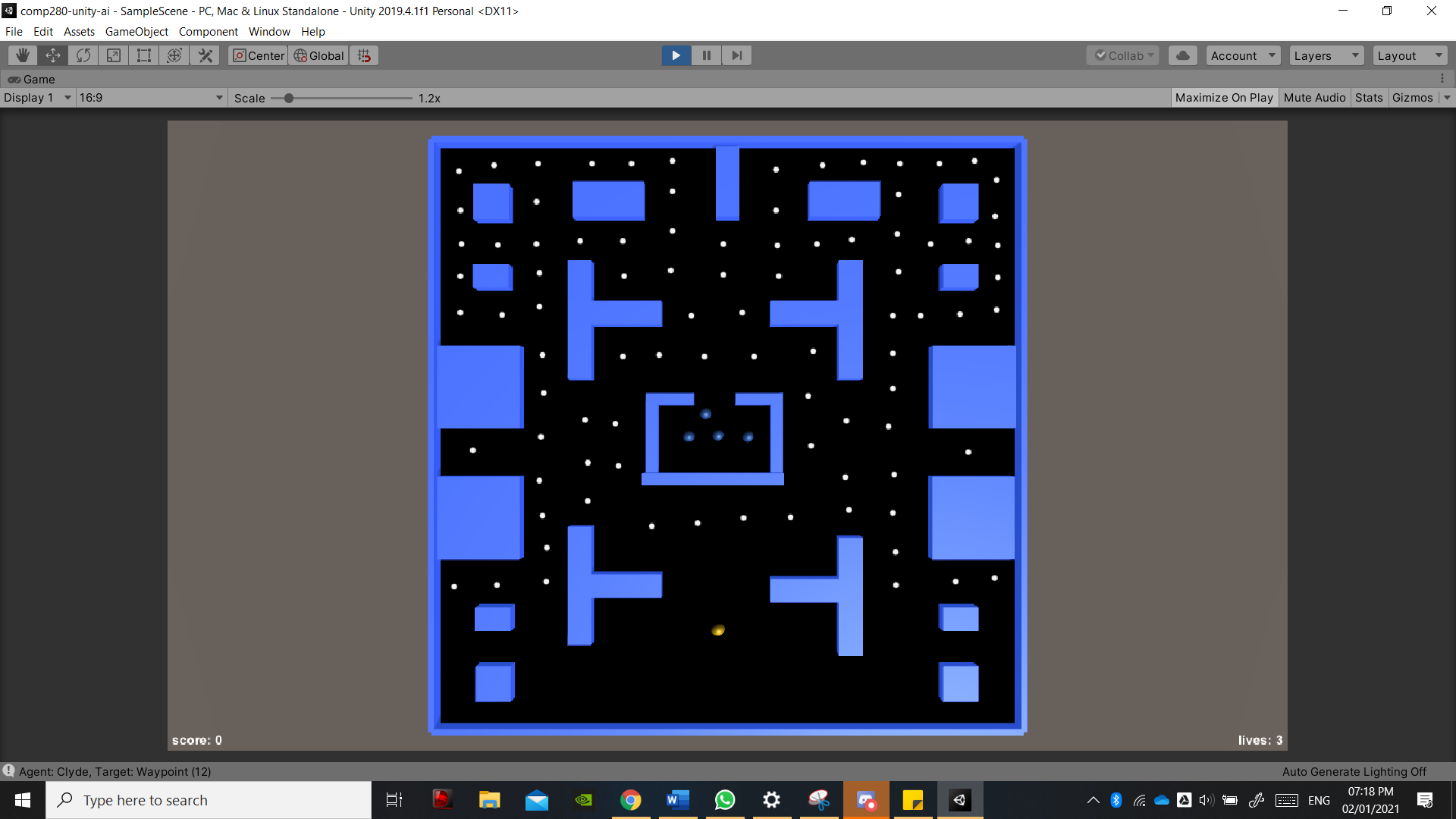
My primary behaviour script gathers the location of every single waypoint and compares it with the location of the player. The waypoint that is currently closest to the player will be targeted by the ghosts. This ensures that the ghosts will travel close to the player, but not to their exact location.

After successfully writing a working ghost AI for COMP280’s Pac-Man behaviour tree task, I noticed my code was rather long, overcomplicated and most likely inefficient. I performed a test run on my code, and although I found it working, each of the ghost’s actions were extremely slow to come out.

**See the repository this project is saved to here:**

[**https://gamesgit.falmouth.ac.uk/users/iw230188/repos/comp280-unity-ai/browse**](https://gamesgit.falmouth.ac.uk/users/iw230188/repos/comp280-unity-ai/browse)

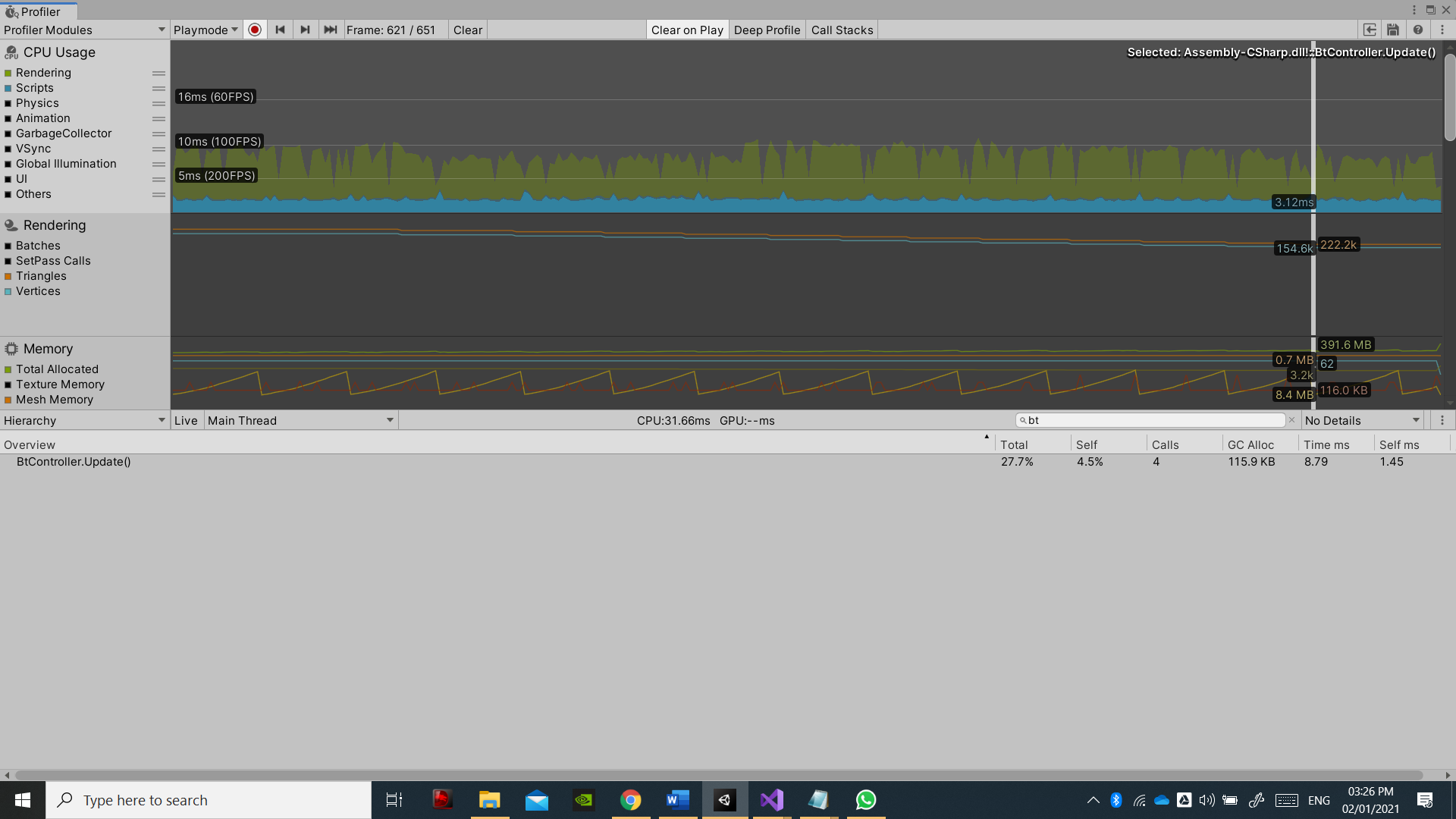
**This repository has an updated version of the AI script that’s analyzed and optimized in this document – it has been updated to provide different AI for each ghost. This isn’t present in the version shown in this essay as it differs too much from the original baseline that was recorded.**

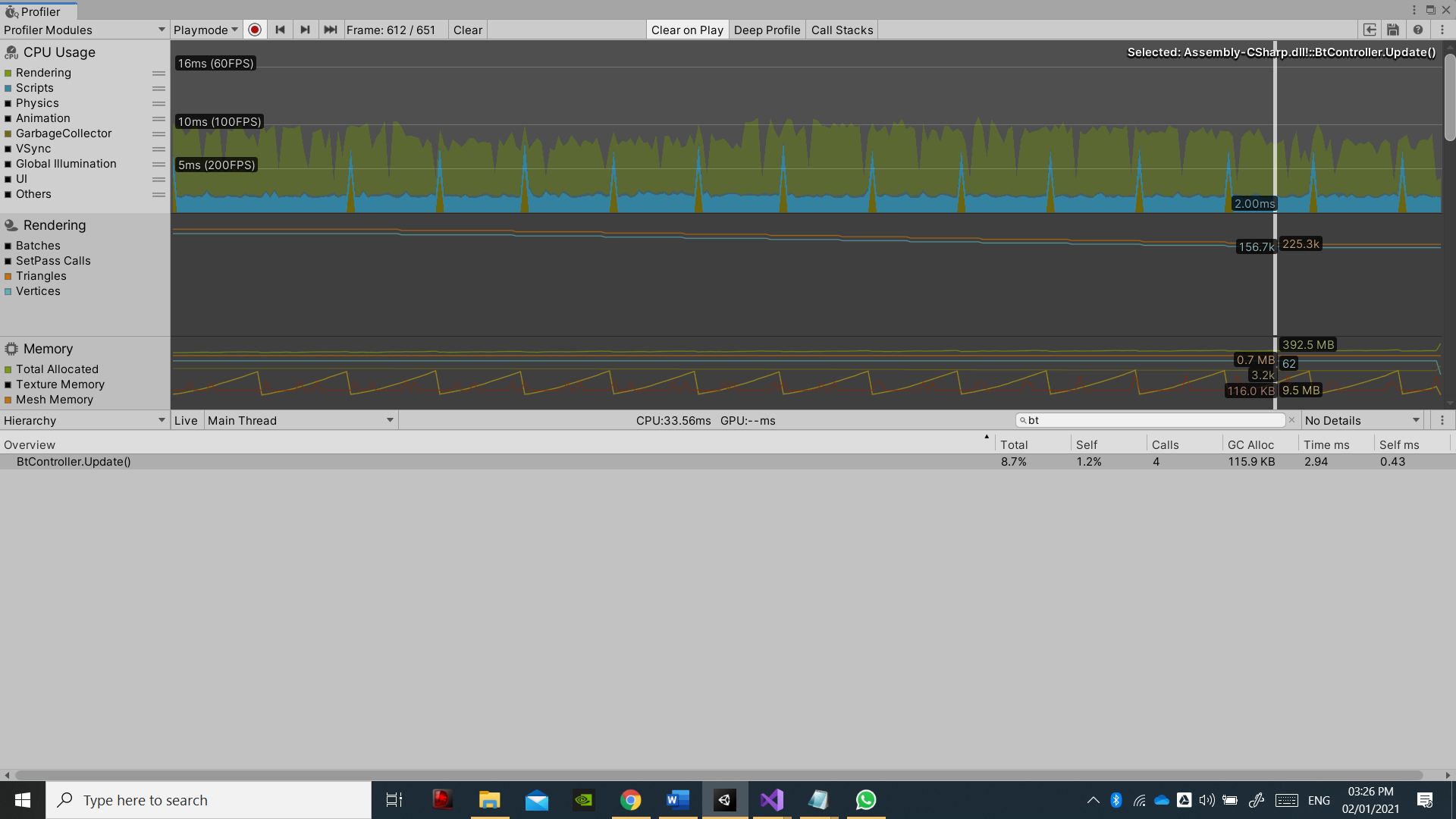


Pictured – the Pac-Man stage detailed above

# Analysing performance – what’s wrong?

To check my hypothesis I opened Unity’s profiler to check how much CPU headroom my behaviour script was using.

As my behaviour script is called by the BtController script’s update() function, this is what will be primarily used to record the performance statistics of the script.



After playing for a short time and gathering performance results in Unity’s Performance Profiler, it is clear to see that there is a noticeable problem with lag spikes caused by my script.

Even in the 300 frames recorded here, the Garbage Collector comes by fairly often, and causes these lag spikes by getting rid of information my script generates. It would be beneficial to remove the need for the Garbage Collector to appear this often, and optimize my script to have less data that needs to be discarded.

Looking at the script itself, in hindsight after viewing this new data, it is apparent that there are far too many variables in use here, as well as too much that the script discards in order to start fresh every loop:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Assertions.Comparers;

public class ClosestToObject : BtNode

{

private string m\_searchingObjects; //The parent of multiple objects the script will take the location of.

private string m\_searchee; //The object intended to be searched for, usually the player

private List<GameObject> distances = new List<GameObject>();

private Vector3 playerLocation;

private List<float> playerDistance = new List<float>();

private GameObject currentClosest;

public GameObject finalClosest;

private bool gotAllChildren = false;

private bool gotAllDistances = true;

private float oldDistance = 999;

private int i = 0;

private int j = 0;

public ClosestToObject(string searchingObjects, string searchee)

{

m\_searchingObjects = searchingObjects;

m\_searchee = searchee;

}

public override NodeState evaluate(Blackboard blackboard)

{

playerLocation = new Vector3();

playerLocation = GameObject.FindGameObjectWithTag(m\_searchee).transform.position;

int children = GameObject.FindGameObjectsWithTag(m\_searchingObjects).Length;

foreach (GameObject targets in GameObject.FindGameObjectsWithTag(m\_searchingObjects))

{

distances.Add(GameObject.FindGameObjectsWithTag(m\_searchingObjects)[i]);

i++;

}

if ((i + 1) >= children)

{

foreach (GameObject distance in distances)

{

float dist = Vector3.Distance(distances[j].transform.position, playerLocation);

playerDistance.Add(dist);

if (oldDistance > playerDistance[j])

{

currentClosest = distances[j];

oldDistance = playerDistance[j];

}

j++;

if ((j + 1) >= distances.Count)

{

finalClosest = currentClosest;

gotAllDistances = true;

if (gotAllDistances == true)

{

Debug.Log(finalClosest);

blackboard.target = finalClosest;

targetclosest();

m\_nodeState = NodeState.SUCCESS;

return m\_nodeState;

}

}

}

}

if (oldDistance == 1000)

{

return NodeState.SUCCESS;

}

else

{

return NodeState.FAILURE;

}

}

void targetclosest()

{

gotAllChildren = false;

finalClosest = null;

distances = null;

i = 0;

j = 0;

distances = new List<GameObject>();

playerDistance = new List<float>();

//gotAllDistances = false;

//gotAllChildren = false;

}

public override string getName()

{

return "ClosestToObject";

}

}

(Also view the script here in your preferred visualizer:)



With 12 different variables, this script uses a lot of information. Many of these variables are lists and GameObjects too (one is a list OF GameObjects), which store a lot of information that will fill up extremely fast. Reducing the amount of these required for the script to run would help tackle this problem

Additionally, the script features many nested loops – at worst featuring an if within an if, which is within a foreach loop that’s also in an if loop. This will no doubt cause some performance problems, as each loop is being checked every frame. Less loops and nested loops are an important change to make for the sake of this script’s efficiency.

At its peak, this script used 27.7% of the CPU available. This game was run on my laptop, which was plugged in during the performance recording, and features an Intel core i5-7300HQ Processor - with each core running at 2.5GHz. While this might not be on the high end of the processing power spectrum, it certainly shouldn’t have this much of the CPU being used by a simple Pac-Man game under any circumstances, let alone from one script out of many being run.

This is also during one of the aforementioned lag spikes – so lowering the amount of these is more important than ever, based on how much CPU power they use.

It should also be a priority to lower the severity of these spikes. No matter how optimized a script in unity is, it has to use the Garbage Collector at some point. While lowering the amount of spikes is a worthy goal, they will still occur, even if it’s rare. Since these spikes are inevitable, reducing the impact they have on performance is a must.

This could be done by changing the type of information the script holds – rather than having the script store the data of an entire GameObject, perhaps just referencing said GameObject through its name or its Tag would be beneficial.

Aside from all these script related efficiency issues, there is a much bigger problem with the toll that rendering takes on the CPU. While the visuals are not affected by me whatsoever, and are simply what have been given to me, it is worth changing this for the sake of the game’s performance.

It's hard to pinpoint exactly what causes so much power to be used up on rendering what appears to be such a simple game. Though upon closer inspection, its visible that every single pill that Pac-Man can eat is a sphere, with its own sphere collider. As I’ve altered the game to use a top-down view, the pills are much further away and less detailed. A quick solution would be to replace the spherical pills with Cubes or sprites, each one offering far less tris needing to be rendered. As well as this, a sphere collider is fairly unnecessary when all that’s being measured is if Pac-Man, a fairly large character, collides with it. Replacing this with a 2D or cube collider would no doubt make the game run faster.

# Fixing These Issues – What performance improvements should take place

After looking long and hard at my script, I realized a large section of it can be removed:

foreach (GameObject targets in GameObject.FindGameObjectsWithTag(m\_searchingObjects))

{

distances.Add(GameObject.FindGameObjectsWithTag(m\_searchingObjects)[i]);

i++;

}

if ((i + 1) >= children)

{

This section of the script is one of the first things declared within the evaluate function. The purpose this section serves is to add every single object with the desired tag to a list of GameObjects (called “distances”), which is then sorted through in this section of the script, with the closest GameObject to the player being saved:

foreach (GameObject distance in distances)

{

float dist = Vector3.Distance(distances[j].transform.position, playerLocation);

playerDistance.Add(dist);

if (oldDistance > playerDistance[j])

{

currentClosest = distances[j];

oldDistance = playerDistance[j];

}

j++;

However, rather than saving each GameObject to a list, it is much faster and easier to simply have the above foreach loop search through each tagged object directly using GameObject.FindGameObjectsWithTag(m\_searchingObjects)[j], rather than reading from the list.

This allows the former shown section of code to be removed entirely, which not only shortens the script and gets rid of a foreach loop, but also means that having a list of GameObjects saved to the script is no longer necessary. This should reduce the severity and quantity of the lag-spikes significantly, as nowhere near as much data is being stored within the script itself now.

There were 25 waypoints being stored in the script by this list. With this gone, performance should be far greater already.

By making this change, the ‘i’ integer is no longer required, as the loop it counts up for has been deleted. ‘j’ is now the only remaining integer, and has been re-named to ‘searchCounter’.

This is a great start, however much more of the script can be removed or changed for the better.

Another pointless variable here is the ‘finalClosest’ GameObject. At the end of the script, once every tag has been searched through and the closest one to the player has been saved to the ‘currentClosest’ variable, the data is then passed on to ‘finalClosest’, which is set as the blackboard’s target. However, why bother passing the information on to another GameObject to be saved in the script, when ‘currentClosest’ by the end of the loop will have the same tagged object saved to it. I’ve changed the script by deleting finalClosest, and renaming currentClosest to just ‘closest’. This is then passed on to the Blackboard at the end of the foreach loop.

At this point the script is looking much more tidy, however there is still the great issue of nested if statements. From further studying, the if ((j + 1) >= distances.Count) statement does not need to be within the foreach loop. This change should allow the script to make less checks and run somewhat smoother,

In this if statement, there is a second if:

gotAllDistances = true;

if (gotAllDistances == true)

This is a fairly obvious one, but it’s pointless to set a variable as true in an if statement, then have if statement only work depending on this change. This was easily resolved by replacing the line gotAllDistances = true; with the contents of the if (gotAllDistances == true) statement. This also removes the need for the gotAllDistances bool entirely.

Finally, another major error was that the sections where SUCCESS and FAILURE are returned are completely inaccessible, as they rely on ‘oldDistance’ being equal to 1000, where it starts off from 999 and can only get smaller from that point. I fixed this to have the final statement at the end of the foreach loop hinge on if the searchCounter (after adding 1) is equal to the length of the retrieved tagged objects. If this is true, then set the target of the blackboard (and therefor AI) to be the closest tagged object to the player. Then return SUCCESS. However if this is false, then an ‘else’ statement will return FAILURE. This ensures the script works stably, and can return its correct status when it needs to.

After making all these changes, and formatting the script on top of some cleanup, this is the final result:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Assertions.Comparers;

public class ClosestToObject : BtNode

{

private string m\_searchingObjects;

private string m\_searchee;

private Vector3 playerLocation;

private List<float> playerDistance = new List<float>();

private GameObject closest;

private int searchCounter = 0;

public ClosestToObject(string searchingObjects, string searchee)

{

m\_searchingObjects = searchingObjects;

m\_searchee = searchee;

}

public override NodeState evaluate(Blackboard blackboard)

{

searchCounter = new int();

float oldDistance = 999;

playerLocation = new Vector3();

playerLocation = GameObject.FindGameObjectWithTag(m\_searchee).transform.position;

int objects = GameObject.FindGameObjectsWithTag(m\_searchingObjects).Length;

foreach (GameObject targets in GameObject.FindGameObjectsWithTag(m\_searchingObjects))

{

playerDistance.Add(Vector3.Distance(GameObject.FindGameObjectsWithTag(m\_searchingObjects)[searchCounter].transform.position, playerLocation));

if (oldDistance > playerDistance[searchCounter])

{

closest = GameObject.FindGameObjectsWithTag(m\_searchingObjects)[searchCounter];

oldDistance = playerDistance[searchCounter];

}

searchCounter++;

}

if ((searchCounter + 1) >= GameObject.FindGameObjectsWithTag(m\_searchingObjects).Length)

{

playerDistance.Clear();

oldDistance = new int();

blackboard.target = closest;

return NodeState.SUCCESS;

}

else

{

return NodeState.SUCCESS;

}

}

public override string getName()

{

return "ClosestToObject";

}

}

Also view this updated script here:



Looking at my new and improved script, it’s evident that it much smaller, being only 60 lines long rather than 95. There is only one nested if statement, which is within a for loop, and only one GameObject variable, which is the ‘closest’ GameObject, is saved.

# Second Analysis – How My Changes Improved Performance

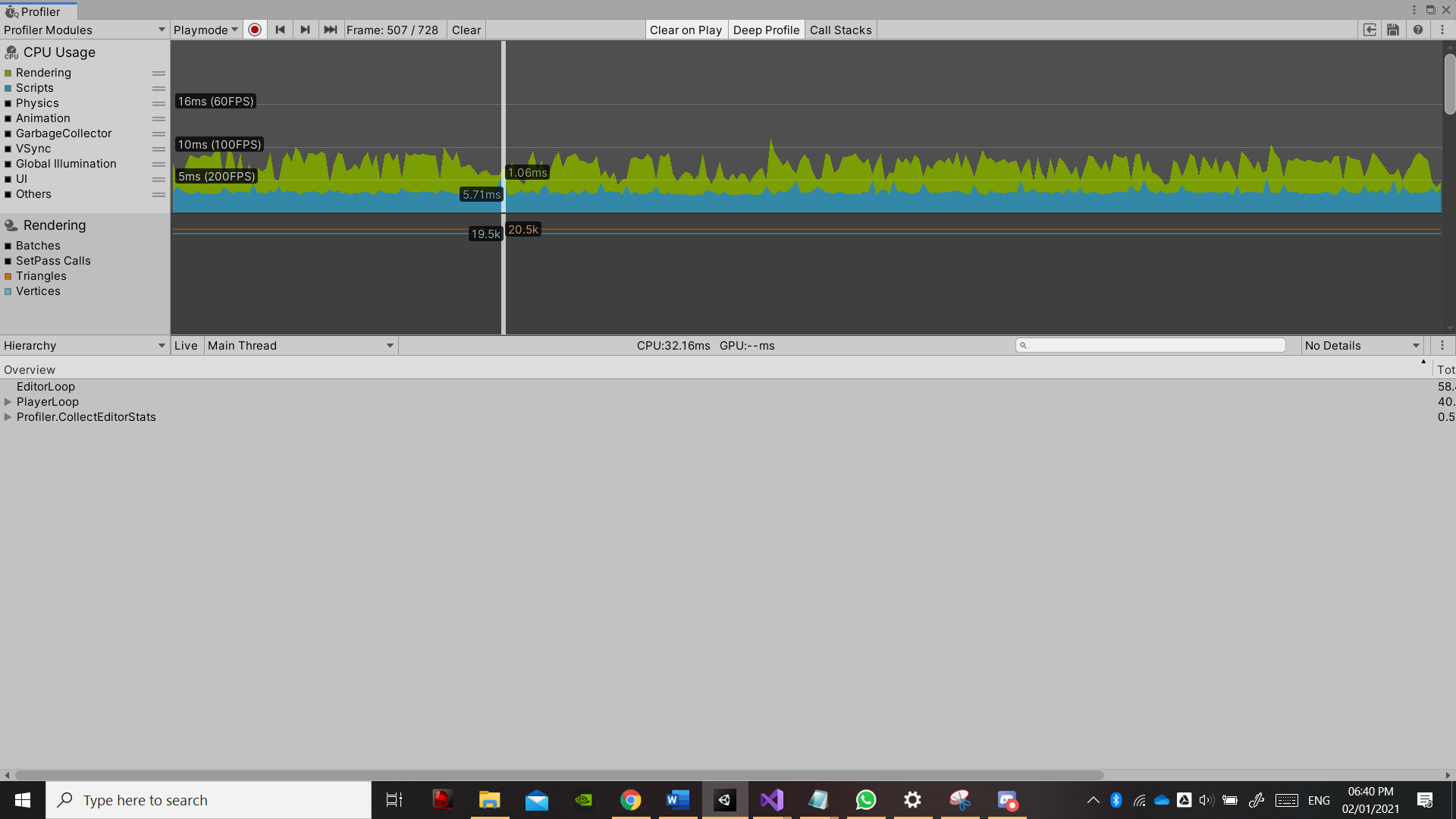
This is the result of my changed script, and the results show! In 300 frames, there are 8 spikes, rather than the 14 present before optimization. In addition, relative to the rendering CPU usage, scripts use far less of the CPU than before.

This recording has taken place before any changes to the rendering, only the script has been changed between the above Performance Profiler screenshots.

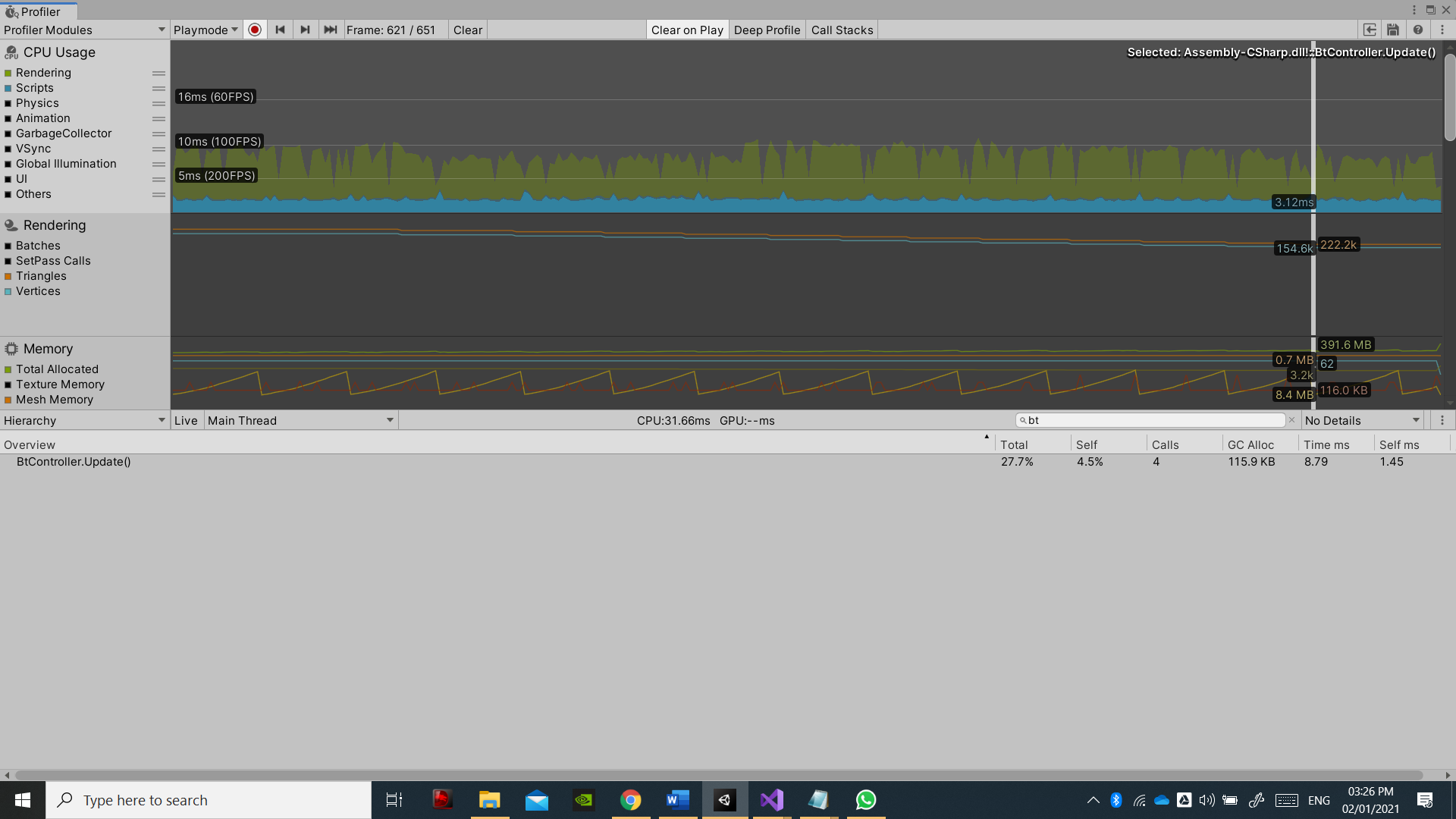
This definitely had the desired effect, as less Garbage Collector lag spikes indicates less information needed to be disposed of. As many variables and lists were removed from my script, this makes sense.

However, something unusual appeared – viewing the above image, the peak CPU usage in a spike caused by the Garbage Collector is actually higher than a few percent in comparison to the previous, unoptimized script.

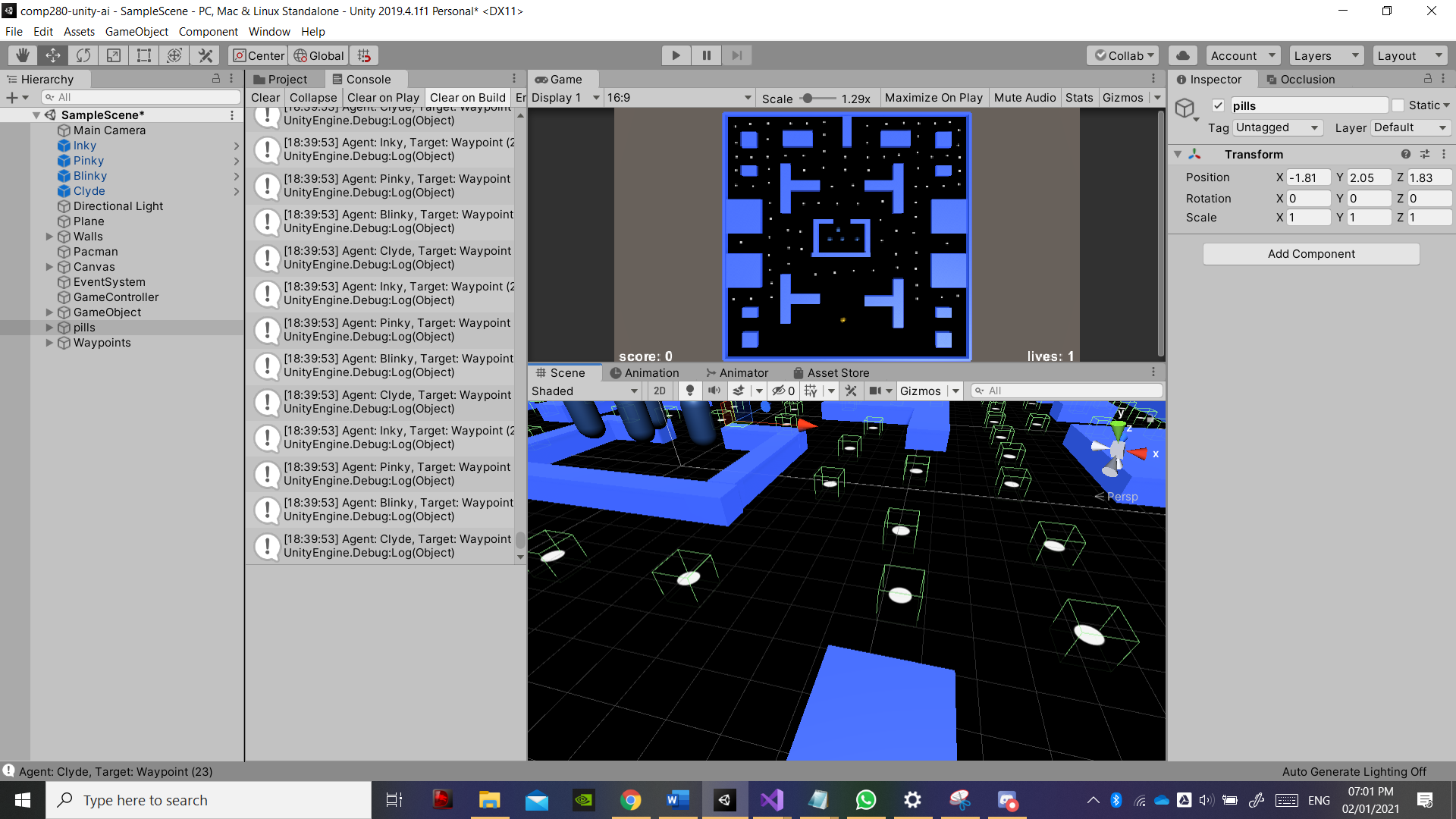
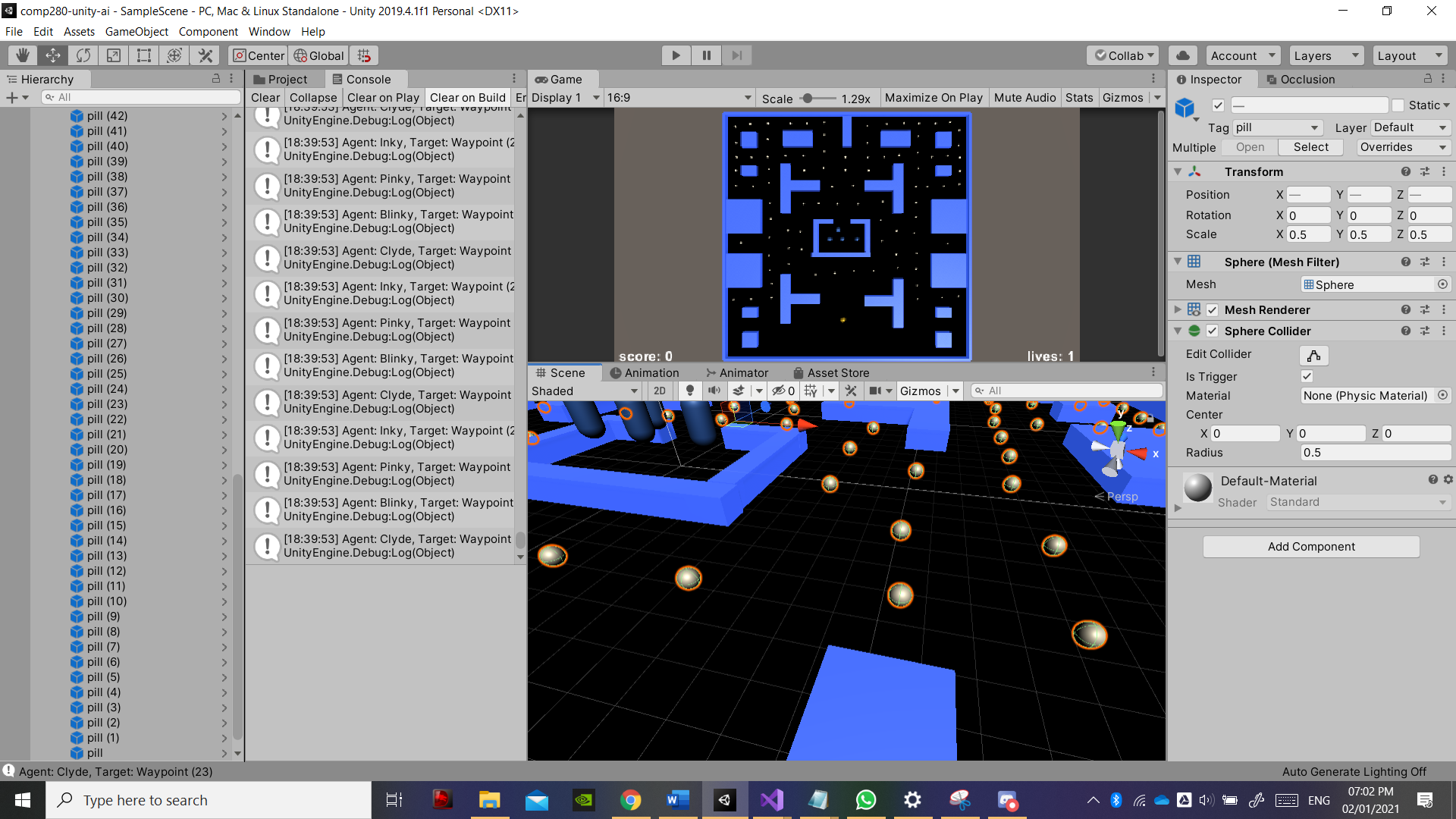
After repeating the test a few times, the CPU usage from each lag spike is around the same on average, so no impact was made on the severity of these spikes.



Pictured above is the Performance Profiler after I changed each pill from a sphere with a sphere collider to a sprite of a circle with box collider. Comparing this to the first profiler screenshot (seen below) the impact on performance is greatly visible, with spikes being lower and an overall smaller impact on the CPU when rendering the game.



This is most likely due to the game instancing a simple sprite, rather than a model with a material that has reflection and lighting data.



This is the visual effect of the change to sprites – and I believe that this has had a positive impact not only on the game’s performance, but also on the game’s visuals and gameplay.

With a bigger square hitbox, the area the player needs to enter to grab the pellet has grown significantly. Now rather than having to come touch them exactly, the player can miss slightly and still eat the pill, leading to less frustration.

Additionally, the circular sprites look more clean, in my opinion. With no lighting data and being one plain colour, they contrast more from the plain black background. Previously the pill’s darker shaded areas from their spherical shape would blend in with the background and obscure them slightly. These new sprites help communicate the shape better and increase visibility.

# Does the data match my prediction?

In many ways yes. There are less lag spikes due to less information needing to be discarded by the Garbage Collector, and less CPU is being used up by rendering by turning each of the pills from spheres with sphere colliders to cubes with cube colliders.

However, the severity of the lag spike stayed the same between scripts, despite being less frequent and having less information to discard.

My belief as to why this occurs is that it’s likely the Garbage Collector works by discarding information based on a set quantity of information each time. Once its accumulated that amount of data, it will discard it. This explains why the Garbage Collector appeared less frequently, as it had more time to build up data until it hit that limit. This also explains why the spikes were equally as severe – an equal amount of data is being discarded for both scripts, even if one takes less time to accumulate it.

To combat this, it would be possible to stop the Garbage Collector, and resume it after a certain period of time. In Pac-Man, it would be suitable to pause it until the next level is being loaded, where between levels all garbage could be disposed of without having an impact on gameplay performance. However, as this version of Pac-Man only has one level and no loading sections, it would be unwise to pause the Garbage Collector.

# Conclusion

I believe I’ve impacted the game’s performance greatly – There are less lag spikes - and while they are still appear sometimes and have a similar severity to before, I’ve explored the options and believe I have optimized the game in the best way possible. There is nothing that can be done about the game’s Garbage Collection in its current state, so should be left intact.