COMP280 - Optimization

Adrian Tofan

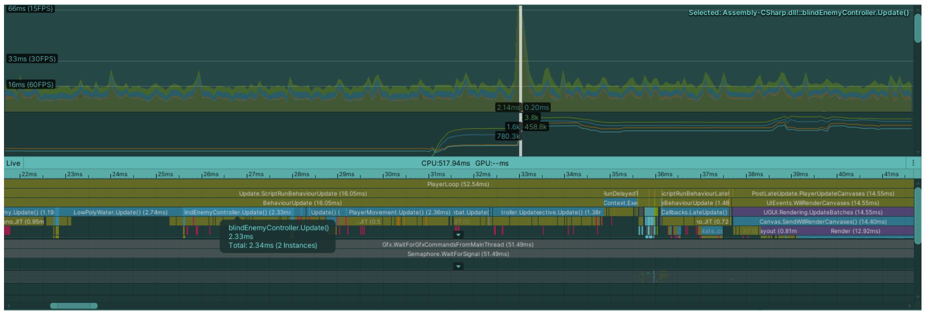
December 2020

1 Introduction

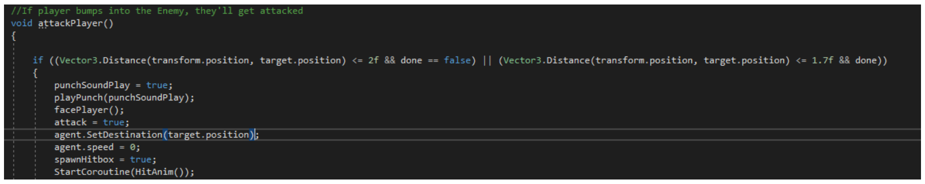
For this worksheet, the project I'll optimize is going to be the first year's team project, *Fuchou*. This project was my first opportunity to work on game AI which is my passion, but unfortunately, I was still really new to Unity and game AI back then and my code was messy and inefficient. This is a perfect time to dive back into that old code and clean it up! Let's begin!

2 Analyzing the profiler and finding solutions

Right from the start, Unity's Profiler helps me notice a huge spike caused by potentially inefficient scripts.

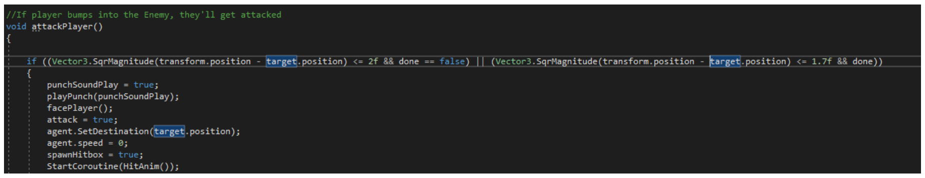

66ms (15FPS)
33ms (30FPS)
16ms (60FPS)
Selected: Assembly-CSharp.dll!::blindEnemyController.Update()
Live
CPU:517.94ms GPU:--ms
PlayerLoop (52.54ms)
Update.ScriptRunBehaviourUpdate (16.05ms)
BehaviourUpdate (16.05ms)
blindEnemyController.Update()
2.33ms
Total: 2.34ms (2 Instances)
Gfx.WaitForGfxCommandsFromMainThread (51.49ms)
Semaphore.WaitForSignal (51.49ms)

The script that was used to control the AI agent I am focusing on is called "blindEnemycontroller" and it appears to affect the performance of the game a lot. One of the first problems I've noticed was the frequent use of **Vector3.Distance** . Even though the overall performance is not greatly affected by a few instances of **Vector3.Distance**, [1] it is recommended to avoid its expensive square root calculations which could potentially create unnecessary work for the CPU by replacing it with the more efficient **Vector3.SqrMagnitude** . In our case, my script made heavy use of the Vector3.Distance.



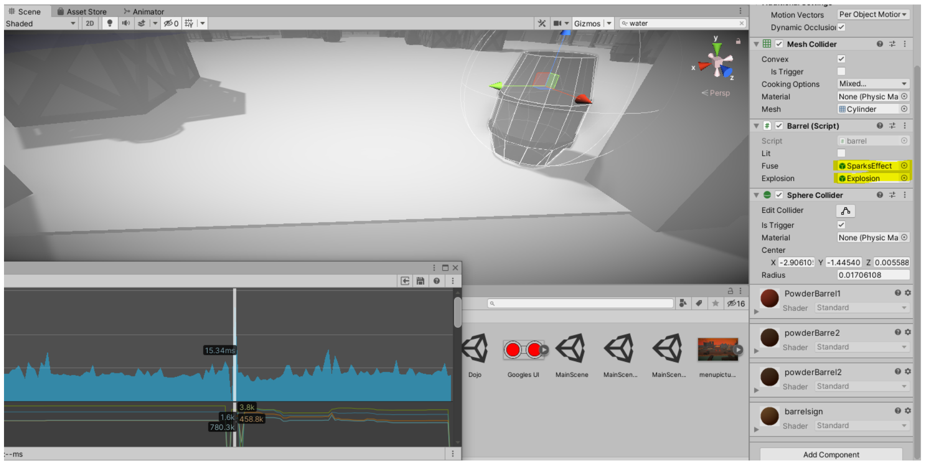
```
//If player bumps into the Enemy, they'll get attacked
void attackPlayer()
{

    if ((Vector3.Distance(transform.position, target.position) <= 2f && done == false) || (Vector3.Distance(transform.position, target.position) <= 1.7f && done))
    {
        punchSoundPlay = true;
        playPunch(punchSoundPlay);
        facePlayer();
        attack = true;
        agent.SetDestination(target.position);
        agent.speed = 0;
        spawnHitbox = true;
        StartCoroutine(HitAnim());
```

This is one of the *if* statements using it that was then modified into a much better version.



```
//If player bumps into the Enemy, they'll get attacked
void attackPlayer()
{

    if ((Vector3.SqrMagnitude(transform.position - target.position) <= 2f && done == false) || (Vector3.SqrMagnitude(transform.position - target.position) <= 1.7f && done))
    {
        punchSoundPlay = true;
        playPunch(punchSoundPlay);
        facePlayer();
        attack = true;
        agent.SetDestination(target.position);
        agent.speed = 0;
        spawnHitbox = true;
        StartCoroutine(HitAnim());
```

Interestingly enough, there was another problem that was caused by another script, linked to the AI agent's code we've been looking at. In order to beat the enemy that used this script, the player has to hit the barrels next to the enemy which would cause an explosion. The script attached to that barrel caused yet another spike because of the barrel script's inefficiency .



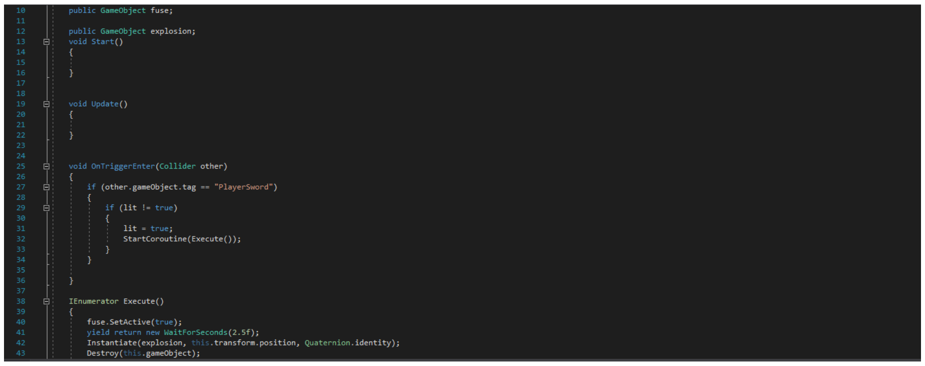
Motion Vectors: Per Object Motion
Dynamic Occlusion: ✓

Mesh Collider
Convex: ✓
Is Trigger:
Cooking Options: Mixed...
Material: None (Physic Ma
Mesh: Cylinder

Barrel (Script)
Script: barrel
Lit:
Fuse: SparksEffect
Explosion: Explosion

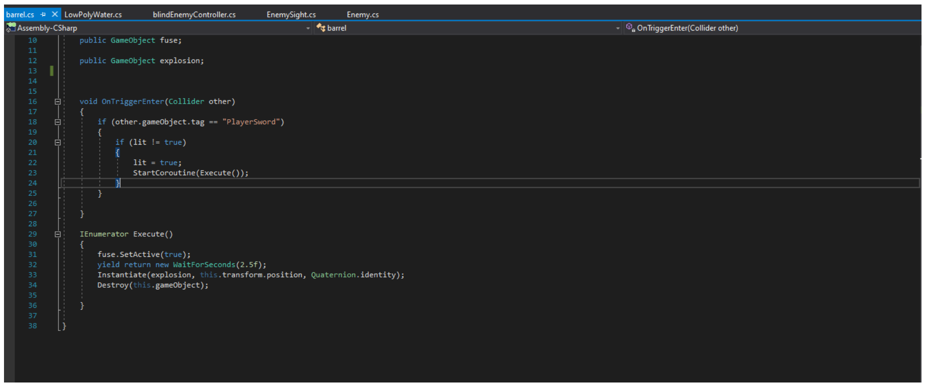
Sphere Collider
Edit Collider
Is Trigger: ✓
Material: None (Physic Ma
Center: X -2.90610 Y -1.44540 Z 0.005588
Radius: 0.01706108

PowderBarrel1 — Shader: Standard
powderBarre2 — Shader: Standard
powderBarrel2 — Shader: Standard
barrelsign — Shader: Standard



```
    public GameObject fuse;

    public GameObject explosion;
    void Start()
    {

    }


    void Update()
    {

    }


    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "PlayerSword")
        {
            if (lit != true)
            {
                lit = true;
                StartCoroutine(Execute());
            }
        }

    }

    IEnumerator Execute()
    {
        fuse.SetActive(true);
        yield return new WaitForSeconds(2.5f);
        Instantiate(explosion, this.transform.position, Quaternion.identity);
        Destroy(this.gameObject);
```

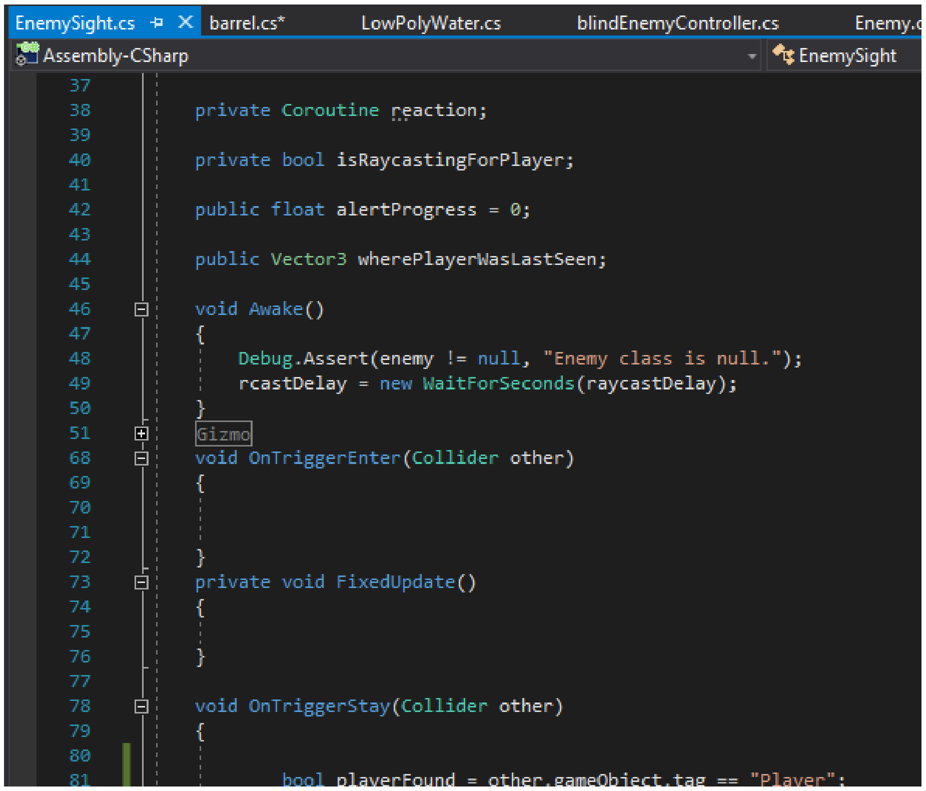
At first glance, these *Start* and *Update* event functions might seem harmless, but *Unity carries out a number of safety checks before calling these functions* [1], therefore affecting the game's performance. Removing them was a solution that saved wasted CPU time.



```
    public GameObject fuse;

    public GameObject explosion;



    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "PlayerSword")
        {
            if (lit != true)
            {
                lit = true;
                StartCoroutine(Execute());
            }
        }

    }

    IEnumerator Execute()
    {
        fuse.SetActive(true);
        yield return new WaitForSeconds(2.5f);
        Instantiate(explosion, this.transform.position, Quaternion.identity);
        Destroy(this.gameObject);

    }

}
```

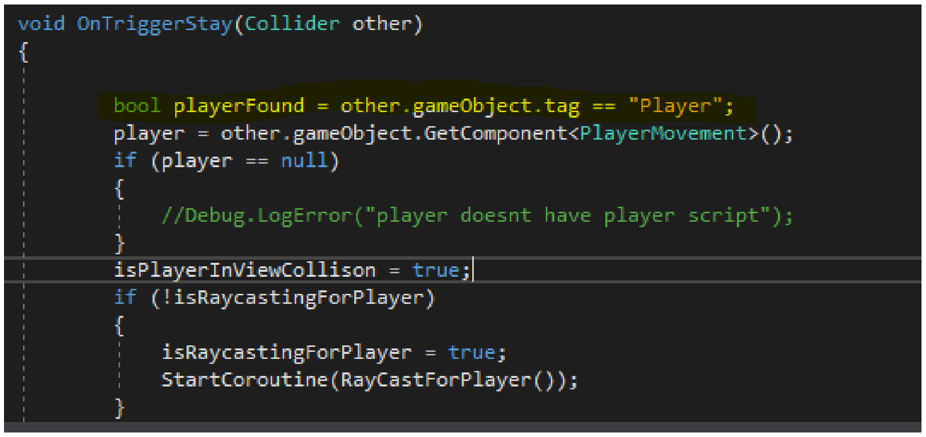
Another problem with the barrel script would be its only "Execute" Coroutine which instantiates an effect and destroys the barrel object. After a while, the barrel would spawn back at the same location anyway, so the best solution for this problem was *Object pooling.* [2] This object pool pattern is a creational design pattern that would use a pool of already-initialized objects rather than initializing and destroying them multiple times. This could *offer a significant performance boost in situations where the cost of initializing a class instance is high and the rate of instantiation and destruction of a class is high – in this case objects can frequently be reused, and each reuse saves a significant amount of time.* [2] In our case, instead of creating and destroying the barrels that would then spawn in the same position anyway, we could activate and deactivate these objects.

A few more issues were also found in the "EnemySight" script which had a few unused methods and also a few lines of code that could potentially generate garbage. First off, let's remove the unused methods.



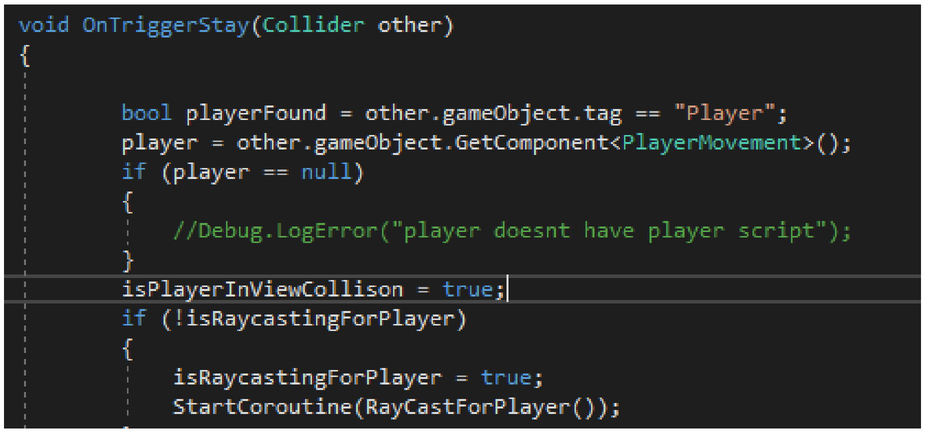
```
    private Coroutine reaction;

    private bool isRaycastingForPlayer;

    public float alertProgress = 0;

    public Vector3 wherePlayerWasLastSeen;

    void Awake()
    {
        Debug.Assert(enemy != null, "Enemy class is null.");
        rcastDelay = new WaitForSeconds(raycastDelay);
    }
    Gizmo
    void OnTriggerEnter(Collider other)
    {


    }
    private void FixedUpdate()
    {

    }

    void OnTriggerStay(Collider other)
    {

            bool playerFound = other.gameObject.tag == "Player";
```

Another problem I've noticed in this script is caused by the call of *GameObject.tag*. This accessor returns string, which means that whenever we call it, it could generate unwanted garbage. Heap allocations can be found within functions like *GameObject.tag* or *GameObject.name*.



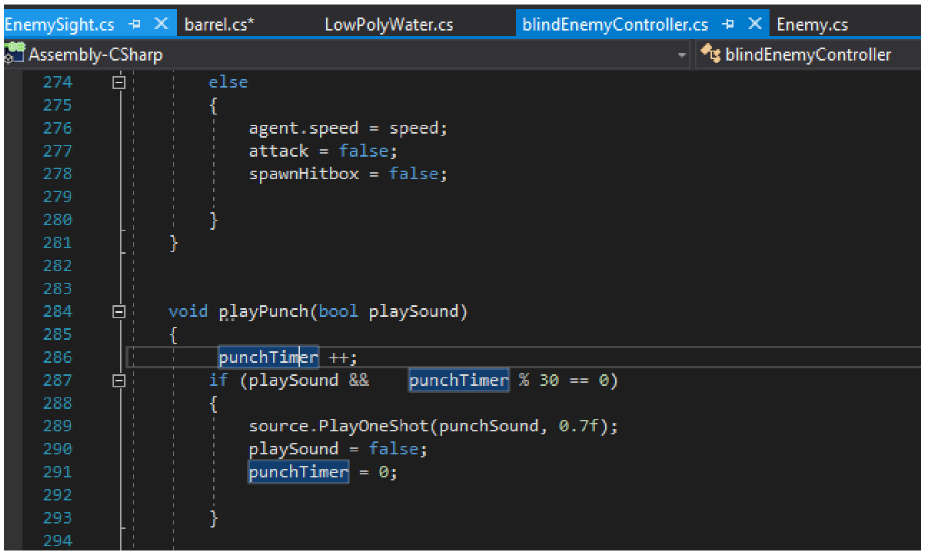
```
void OnTriggerStay(Collider other)
{

        bool playerFound = other.gameObject.tag == "Player";
        player = other.gameObject.GetComponent<PlayerMovement>();
        if (player == null)
        {
            //Debug.LogError("player doesnt have player script");
        }
        isPlayerInViewCollison = true;
        if (!isRaycastingForPlayer)
        {
            isRaycastingForPlayer = true;
            StartCoroutine(RayCastForPlayer());
        }
```

To solve this, we can use *GameObject.CompareTag()* instead.

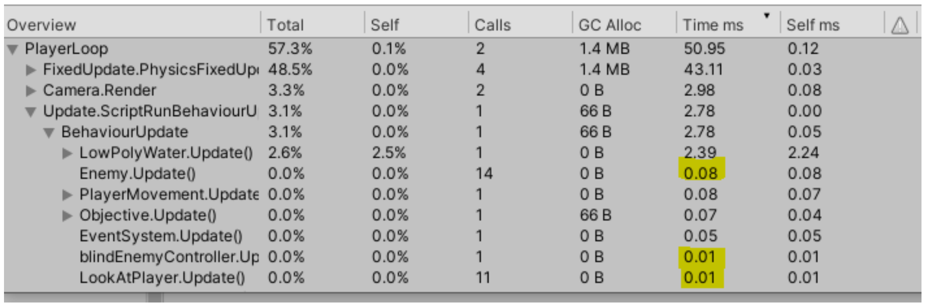


```
void OnTriggerStay(Collider other)
{

        bool playerFound = other.gameObject.tag == "Player";
        player = other.gameObject.GetComponent<PlayerMovement>();
        if (player == null)
        {
            //Debug.LogError("player doesnt have player script");
        }
        isPlayerInViewCollison = true;
        if (!isRaycastingForPlayer)
        {
            isRaycastingForPlayer = true;
            StartCoroutine(RayCastForPlayer());
```

Looking back at the "blindEnemyController" script, I've noticed that back then I didn't know that I could use *Time.deltaTime* for timers. This script makes heavy use of timers that are integers initialized with 0 every time, every single one of which increases by 1 every frame. To check if a certain amount of time has passed, I used the mod operation. This way of creating timers was really inefficient and now I can remove most of them and replace the main ones with += Time.deltaTime or -=Time.deltaTime depending on the case. One good example is the use of the integer called "punchTimer" that was used to play a punching sound for a certain amount of time.



```
            else
            {
                agent.speed = speed;
                attack = false;
                spawnHitbox = false;

            }
        }


        void playPunch(bool playSound)
        {
            punchTimer ++;
            if (playSound &&    punchTimer % 30 == 0)
            {
                source.PlayOneShot(punchSound, 0.7f);
                playSound = false;
                punchTimer = 0;

            }

```



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	57.3%	0.1%	2	1.4 MB	50.95	0.12
▶ FixedUpdate.PhysicsFixedUp	48.5%	0.0%	4	1.4 MB	43.11	0.03
▶ Camera.Render	3.3%	0.0%	2	0 B	2.98	0.08
▼ Update.ScriptRunBehaviourU	3.1%	0.0%	1	66 B	2.78	0.00
▼ BehaviourUpdate	3.1%	0.0%	1	66 B	2.78	0.05
▶ LowPolyWater.Update()	2.6%	2.5%	1	0 B	2.39	2.24
Enemy.Update()	0.0%	0.0%	14	0 B	0.08	0.08
▶ PlayerMovement.Update	0.0%	0.0%	1	0 B	0.08	0.07
▶ Objective.Update()	0.0%	0.0%	1	66 B	0.07	0.04
EventSystem.Update()	0.0%	0.0%	1	0 B	0.05	0.05
blindEnemyController.Up	0.0%	0.0%	1	0 B	0.01	0.01
LookAtPlayer.Update()	0.0%	0.0%	11	0 B	0.01	0.01

3 Conclusion

As we can see, the performance has been drastically improved, although there are still spikes that pop up at certain times, potentially decreasing the FPS. There is still room for improvement but this project has helped me understand the importance of having optimized, efficient code and how much certain habits that seem harmless could potentially affect the performance of my projects.

References

[1] Unity Technologies. Optimizing scripts in unity games.

[2] Wikipedia the free encyclopedia. Object pool pattern.