SWE585 Term Project

Game Development: Armstrong Strongarm

Group Members:
Fatma Betül Güreş,
Deniz Dikbıyık

Date: 05.06.2022

Instructor: Atay Özgövde

Introduction

The game is a First Person Shooter (FPS) game, where the astronaut must shoot alien enemies.
You as a player: are an astronaut with a costume that offers you gravity control ability (under your feet there is a magnetic system) and you must withstand alien attacks to escape.
Them: Element bender powerful aliens.

Description

Armstrong Strongarm is a fast-paced first person shooter. Set on Planet EFAW (Earth, Fire, Air, Water) outdoor environments specific to the element bender aliens. The world might be thought of as a parallel universe to Earth, but with four types of aliens who live in harmony as they need each other and each season (spring, summer, fall, winter) another species is ruling. They have some fancier technology and the ability to control elements.
The player must not only eliminate the attack of the aliens, but must strive to maintain battery level in the suit, which is rechargeable when you reach batteries.
The aliens don't understand your purpose and see you as a thief, so they attack you. They will be using the NavMesh agent and Unity AI elements to chase you. As you are an astronaut with a costume that offers you gravity control ability this power will offer flexibility during fights.

Game Mechanics

Movement of the Player
The player is moving around in the game environment, holding a series of guns and being able to change guns and zoom in or zoom out using the lense of the guns.

Shooting
There are different types of guns which include different numbers of ammos inside. To kill the enemies, shooting is done by the player.

Changing Weapon
The other types of guns are changed to get another charged one.

Collecting Battery
The batteries are spread around in the game environment, so the player is able to reach them and collect them to increase the ability to survive.

Collecting Ammo
The guns are filled with different types of ammos, so ammos around the game environment are collectible.

Controlling Gravity
The astronaut, who is the game player, is able to change the gravity in some situations in the game. If there is a platform too high to jump, the astronaut can change gravity values to get there.

Enemies are Chasing
The enemies which are defined above are coming towards the player, when they realize the player is near them (if within chase range) or if the player shoots them (and they are still alive to chase).

Enemies are Attacking

Enemies are attacking the game player to decrease survival ability. Blood is seen on the screen when there is an attack by the enemy and health points of the player decreases.

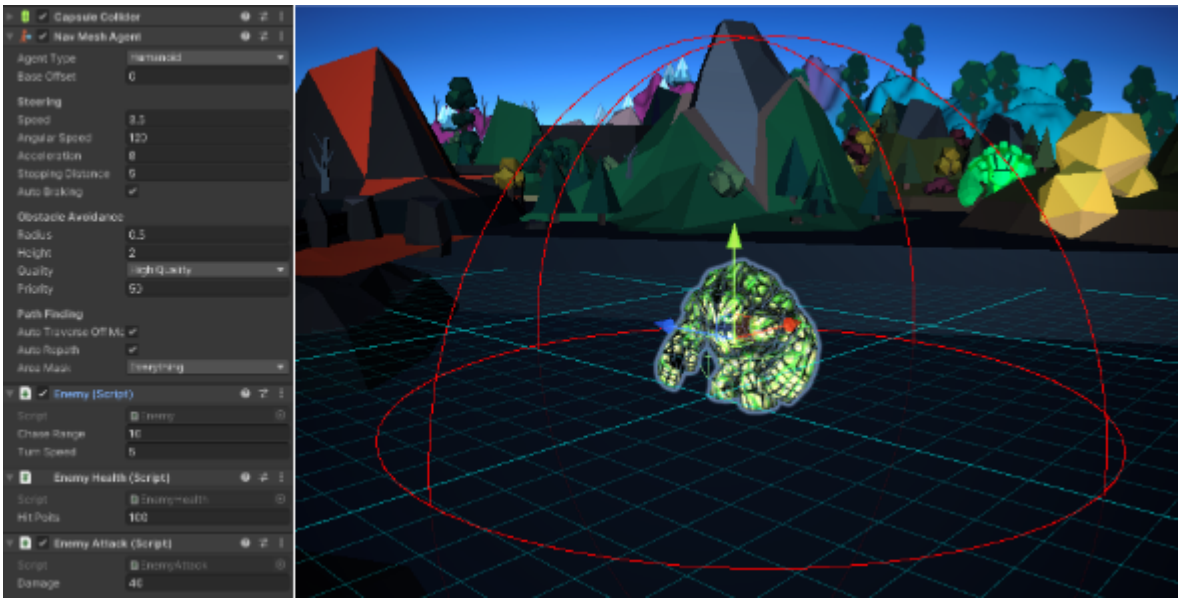
Asset List

Guns
Batteries
Ammo
Sky
Particles
Enemies: Min Legion Rock Golem in Asset Store
Game Environment: Fantasy_Environment Pack in Asset Store

Summary of the Implementation Process

First of all, we wanted to improve our knowledge about Unity and took this course on Udemy. Complete C# Unity Game Developer 3D (https://www.udemy.com/course/unitycourse2/). We have both watched the course, especially the section related to 1st Person Shooter development. The course was about a zombie shooter, but we have applied the learned skills to develop the game for the things we have proposed in Project Phase 1, Armstrong Strongarm.

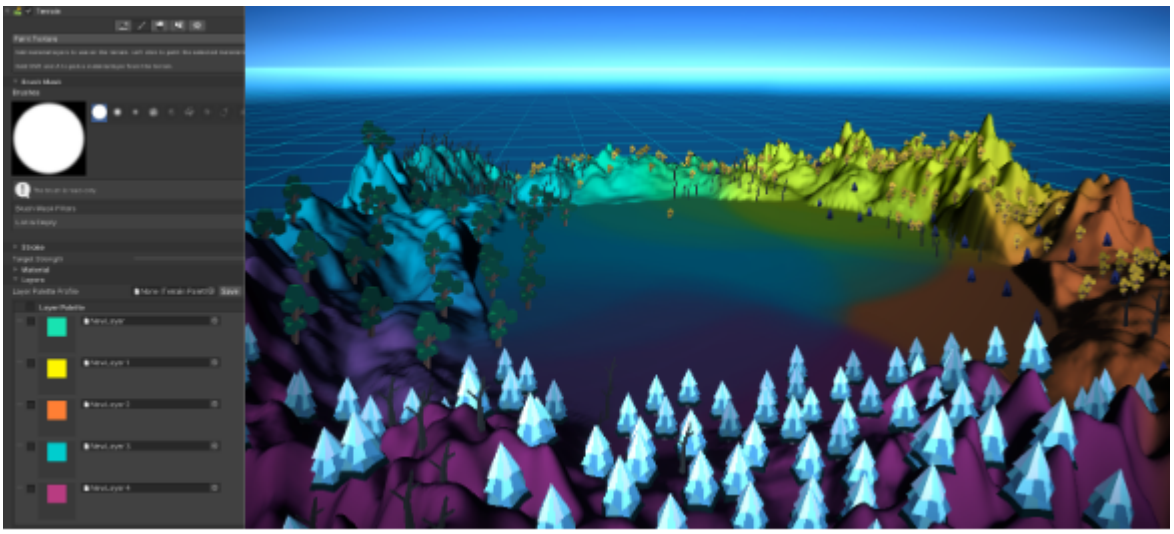
We added red wireframe spheres **-gizmos-** to the enemies which show the chase range in scene view. Chase range is in fact the distance, if triggered by the player, the enemy starts walking towards the player.



We use a raycast system to shoot the enemies. Raycasting can be thought of an invisible ray, of a certain length, coming out of the camera (field of view of the player). This invisible ray is looking for collieders. In our case, if the shooting raycast hits the enemy collider, the enemy will get damage.

After shooting, particle animation is shown to make shooting more visible. One particle appears at the tip of the weapon, and the other particle is instantiated at where the raycast system hits. Players can pick up ammo and batteries which increase ammo number or the lightning.

Terrain: We have used Unity's terrain tools to define the boundaries of the game world. We didn't want our player to go out of the game-zone, therefore made use of high altitudes at the surroundings. We have created materials with colors matching our game world and have also added lowpoly trees (once again added with the terrain tool, not added gameobjects one by one).
Pro-Grids and Pro-Builder: Both of them are Unity packages used for prototyping and creating level design. Pro-builder is like a simplified modeling tool that can be accessed within the game engine. Progrids, as the name suggests, creates helper grids for the user, and can be used together with pro-grids. We have experimented with these tools when creating the level design. Also we have created simple assets, like different ammos that can be picked up to recharge our weapon.



So, the main game is based on:

- Enemies that attack when provoked
- Shooting function
- Flashlight mechanic with recharging
- Ammo count
- Ammo pickups
- Weapons we can cycle through
- Zoom In-Out with some weapons

Minimum Viable Product (MVP):
The core events of the game are listed below.

- First Person Camera movement
- Raycasting to shoot
- Enemies move and attack AI
- Health and damage system
- Death/game over
- Weapon switch
- Ammo pickup
- Flashlight pickup

Implementation Process

First Person Controller:
When we created the project, we downloaded Unity Standard Assets. We did not add sample scenes, 2D things, physics materials and vehicles. After that, we created one plane. From the standard assets, we added a rigidbody fps controller as a first person controller. The name is changed as the player. We deleted the main camera because the one under player was enough for us.

NavMesh Agent for AI:
This is a helpful AI feature of Unity. We are using this for the enemies to come towards the player and be able to walk around the obstacles. Using the parameters we define the platforms that the enemies are able to jump on and off. We started implementing NavMesh by creating a 3D capsule. It was

really useful to see the main functions. We made the capsule collider true and added a NavMesh agent. This is the enemy which moves around.

Chase Range:
We made the enemy prefab and worked on the chase range. Chase range is about following the player and the work we did here was about how close will the enemy be to the player. The player is the distance target for the enemy, so the values are important here. We used OnDrawGizmosSelected to make the surrounding chasing ability. The following method is in Enemy.cs.

```
void OnDrawGizmosSelected()
  {
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position, chaseRange);
  }
```

Enemy AI to Attack If Provoked:
Enemies also follow the player when they are shot. They follow if the player is in the visible area to them, they follow and attack if close enough. So, the functionality is based on engaging target related to is provoked boolean and make is provoked boolean true if close enough, then follow. In the engage situation -meaning that the player has shot and provoked the enemy- the enemy starts chasing the player no matter the distance. Once close enough to attack and give damage, it enters the attack state. In the animator, attack animation starts to play.

Gun to the Player:
We have 3 guns and add them as child objects to the player's camera, by dragging. By changing the weapon model's scale, position, color and rotation, we made all weapons aim at the center of the screen, as if our first person character is holding the weapon in his hands. After obtaining the desired looks, we applied the changes to prefab. To see the target point reticle, we added a UI canvas and placed the gun reticle image at the center, by adding an image to the canvas and attaching the correct image to the sprite renderer.

Raycasting:
Raycasting is based on origin, ray and collision. We have a weapon script for it. By this functionality, we can focus on the target by a line and shoot.

Enemy Health and Damage:
We created an enemy health script for this and called on weapon to decrease the health of the enemy when it is shot. With the values we have attached to the enemy health and weapon's shooting damage, the enemy can die when 3 shots are done.

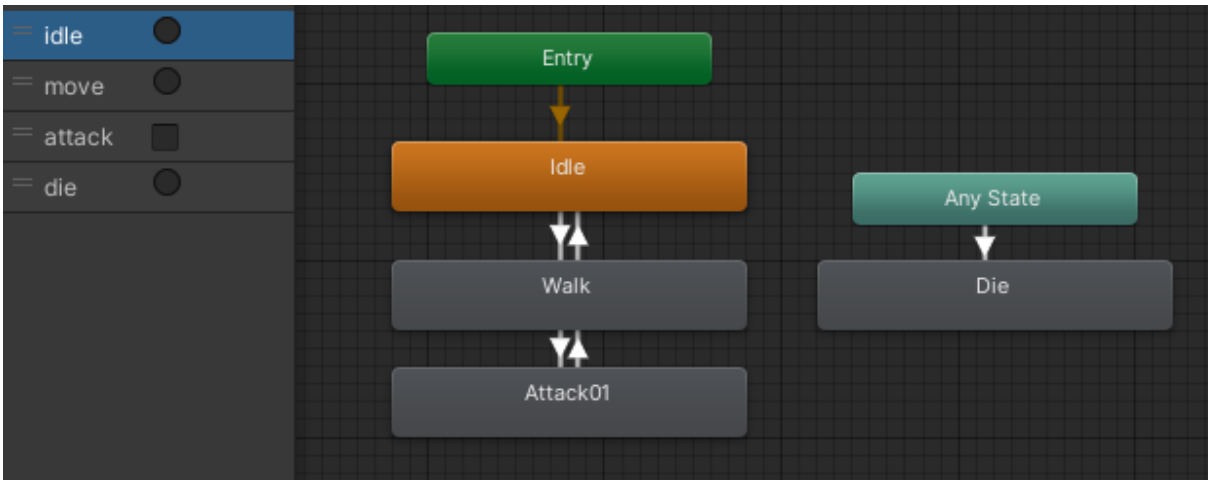
Muzzle Flash:
We added this by adding the particle system from effects option. This is for the shooting effect. It can be found as VFX under prefabs. Looping and play on awake should be made false to call this only if shooting is done. The simulation space is world, and we changed the duration, lifetime, speed, and rate over time. We dragged this to the gun. We triggered this on the weapon scripts and added process raycast and play muzzle flash methods. By adding the material to the code, we called this easily and finally made the visuals more realistic.

Shooting Effect:

We use explosion here from the standard assets. We made it prefab and designed later. The code related to it is process raycast. It is instantiated there with quaternion look rotation and destroy it after 1 second.

Animations:

There are animations and transitions. From animation component, we assign animations to the game objects through **animator** controller. From animator controller, we can arrange the animations and transitions (state machine).

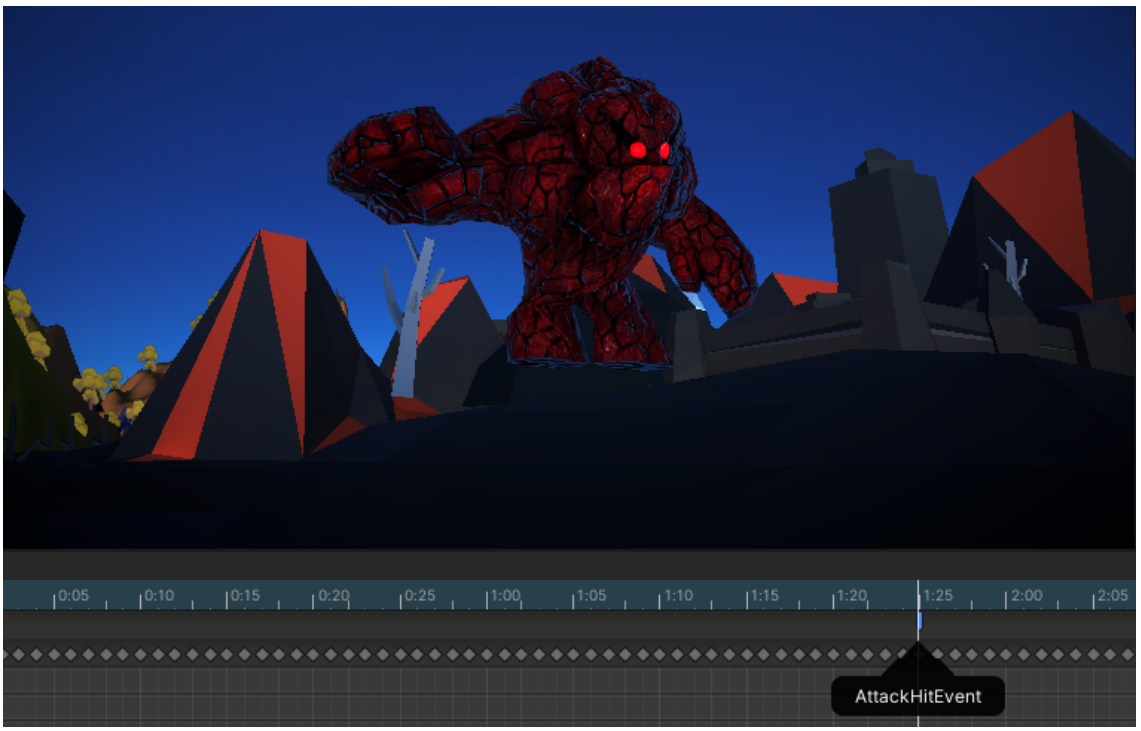


There are Entry, Idle-Move (Walk)-Attack states and AnyState-Die. These states are called according to triggers and booleans. Die can be called from any state because it can occur any time. We added animator component to the enemy and created animator controller. By creating states, we added the ones mentioned above. Transitions are also added accordingly. The settings on the right side of the editor are important to make transitions smooth. We made exit times around 1 second. We added animator controller to the enemy. Normally, animations can be created by recording, but we had very useful animations coming with our enemy assets. To the animation states above, we have attached the animations of the enemy for: idle, walk, attack and die. All enemies use the same animator, as they have the exact same behavior (only the colors and materials are different among the element bender aliens). The transitioning among the animation states are controlled with the triggers and booleans. In our game logic, once the player enters the chase range, the move trigger is fired from code. Similar to triggers, we also have booleans “attack”, which is set to true and false according to the desired gamelogic.

Enemy.Animator.SetTrigger("move");

Enemy.Animator.SetBool("attack", true);

Inside the attack animation there is an event called “**AttackHitEvent**”. At this frame of the animation (when the enemy is punching the player) the player gets health damage.



Player Health:

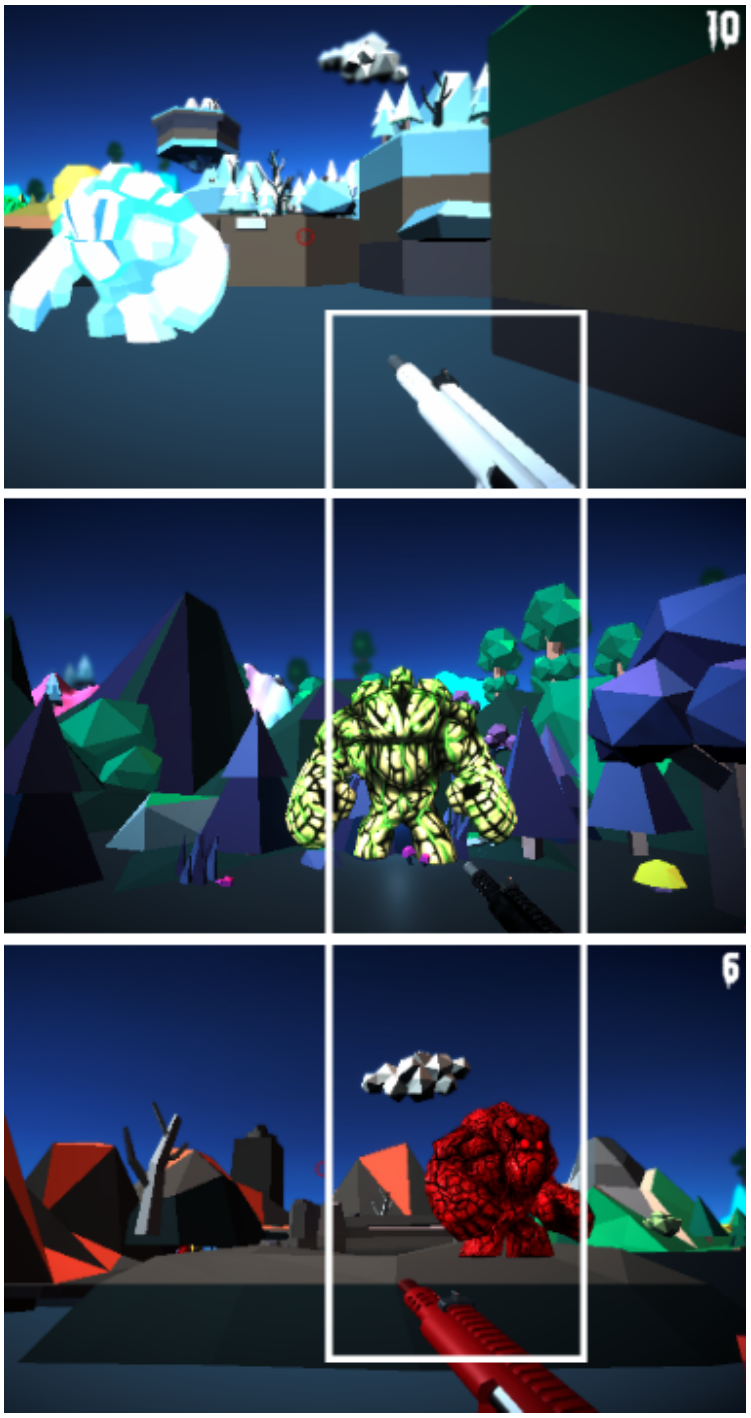
This is also a script similar to enemy health. We call this on enemy attack code with component of target. Because it is hard to call it with a target, we called from player health with an object in the enemy attack.

User Interfaces:

To create user interfaces, we have made use of Unity Canvases. We used more than one and activated/deactivated them, depending on the need of the game. For instance, we created the game over the user interface and included a play again button. In order to make this button functional, we created a public method that reloads the game (by reloading the scene). Similarly, to exit the game, we created another public method and attached it to the quit button.

Weapon System:

We have developed a relatively complex weapon system for our game. We did not only attach one gun, but three, from which the user can choose what they want. To change among the weapons, the user can scroll the mouse wheel or press on 1, 2 or 3 from the keyboard. Visually, we have made differences regarding color and scale, however we wanted to have some functional differences as well. For instance we wrote a script (WeaponZoom) for zooming in and out- and only attached it to two weapons. By toggling (from script) the Field Of View (FOV) variable of the camera, we obtained a basic zoom effect. To develop this feature even further, we wanted to make the zoomed-in version less sensitive so that the player can smoothly observe the distance and make more precise shots.



Another complex feature regarding the weapon system is the fact that, for filling each gun, we require another type of bullet. In the picture above, we called our red weapon-**Shotgun**, the white one is **Carbine** and the black one is **Pistol**. To shoot with Shotgun, we need **Bullets**; for Carbine we need **Shells**, and for the pistol, we need **Rockets**. In order to differentiate these features, we have created an enum script (which is not of type MonoBehaviour).

```
public enum AmmoType
{
   Bullets,
   Shells,
   Rockets
}
```

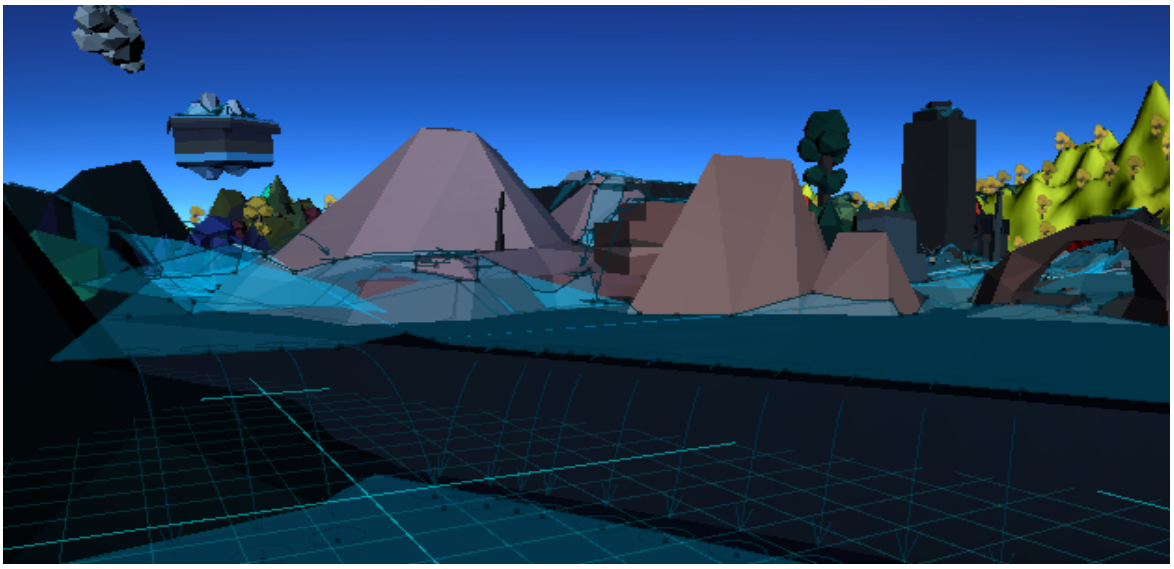
Gravity:

To offer gravity control to our player, we created inputs, such that when the player presses "G" key, the Physics.gravity is being multiplied with -1 (therefore changing direction). To activate this gravity switch, the player must realize a jump- which will add force, and will let player move in the chosen gravity direction.

Technical Challenges

NavMesh:

Armstrong Strongarm is a 1st person shooter, where all the enemies are using the unity.ai library. These are moving on their own, with the component NavMesh agent. To see the objects that an enemy can move on, the **Bake** can be checked. Static should be selected for all the surface gameobjects that the player can move on. Changing agent radius and max slope are important to have more realistic movements. We have done fine tuning by playing with the values, until the baked area made sense for our game environment. At this point, we started to write our enemy AI script. We gave a destination which is set according to the target position that allows us to make enemies follow the player. On the main camera of the player, the head bob is closed to remove unnecessary stucks. There was a problem with physics material like a wall, we changed the numbers to 0 and added the player's capsule collider, then made the friction minimum. So, the problem is solved.



Raycast:

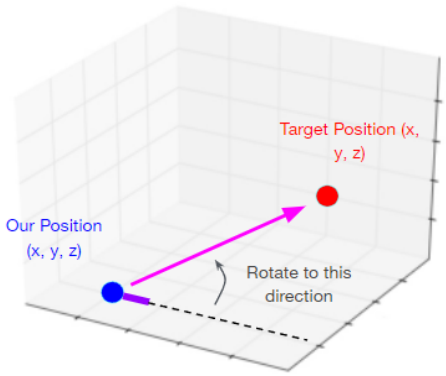
The Raycasting system works like an invisible ray of a certain length, that is shot from a starting point (First Person Camera) towards a certain direction (foreward). This is especially useful for shooting mechanics. If the invisible ray of the raycasting system collides with colliders in the game scene, certain actions will be done (damaging enemies). The following code can be found in Weapon.cs.

```
void ProcessRaycast()
  {
    RaycastHit hit;

     if (Physics.Raycast(FPCamera.transform.position, FPCamera.transform.forward, out hit, range))
    {
      CreateHitImpact(hit);
      EnemyHealth target = hit.transform.GetComponent<EnemyHealth>();
      if (target == null) return;
      target.TakeDamage(damage);
    }
    else
    {
      return;
    }
  }
```

Enemies Facing the Target When Moving:

When creating the chasing functionality of the enemies, first we have used placeholders (the capsule gameobject in unity). Therefore, we did not worry about the faces of the enemies to look towards the player. However, once we have placed the character assets, we have seen the necessity of this functionality. To solve this challenge, we have made use of vectors. When subtracting the position of the enemy from the position of the player, we have calculated the vector between them. To allocate a direction to the enemy, we do not need the entire vector- unit vector is enough. By using the quaternion function of Unity (which is a way of defining orientation in 3D space, by using calculation methods that imply an imaginary 4D space) we have totated the orientation of the enemy, until it was facing at the direction of the unit vector we calculated (described above).



Gravity:

An important feature for our game was the gravity control ability. This has differentiated our game from other 1st person shooter games. However, the gravity control implementation was a little bit of a challenging process. We have observed that the rigidbody2D is more flexible in this aspect in comparison to the rigidbody for 3D gameobjects. **Rigidbody2D.gravityScale** is only available for rigidbody2D.

A solution we have thought of was to go to player settings and play with gravity values from there. If we went for this approach, this time the gravity would have changed for all gameobjects and not only our player- which made us worry at first. However, the only gameobject that would get affected by gravity is our player, as the enemies are moving around as NavMeshAgents and the rest of the game objects are static game environment assets.

```
void Update(){
   if(Input.GetKeyDown(KeyCode.G))
   {
      Physics.gravity *= -1;
    }
```

Coroutine on Shooting:

For the smooth change between shootings, we use coroutines to make the player wait before the next shot. They are not shooting automatically. Instead, they need to give the input each time. In order to obtain a delay in time and not directly do the operations in the speed offered by unity frame, we made use of "Coroutine". When creating coroutines there are 3 main steps that are different from a standard method. Firstly, when defining the coroutine, we use the keyword IEnumerator. To obtain the delay in time that the game requires, we can type "yield return new WaitForSeconds" and then type in the time delay we want to achieve. Finally, when calling the coroutine, we must say StartCoroutine and type in the name of it within parentheses.

```
IEnumerator Shoot()
{  canShoot = false;
   if (ammoSlot.GetCurrentAmmo(ammoType)>0)
   {
      PlayMuzzleFlash();
      ProcessRaycast();
      ammoSlot.ReduceCurrentAmmo(ammoType);
   }
   yield return new WaitForSeconds (timeBetweenShots);
   canShoot = true;
}

…..

if (Input.GetMouseButtonDown(0) && canShoot==true)
   {
      StartCoroutine(Shoot());
   }
```

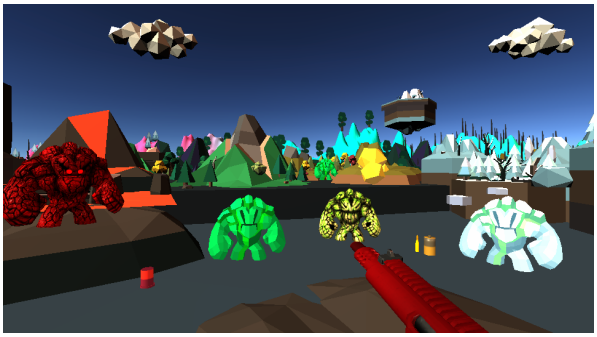
Libraries

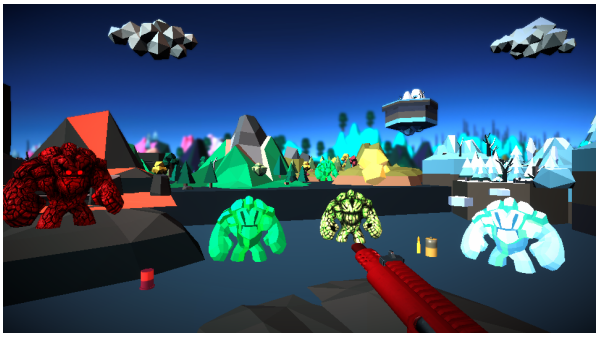
Text mesh pro:

For using the canvas system with text that is suitable with the game theme, we have visited the site: “dafont.com”. Here we have chosen a font style that we like, and is suitable for the game style, among the fonts that were free for personal use. After this download, we went back to Unity and used the **Text Mesh Pro Package** such that we generated the font atlas. After that, we were able to select this front from the text we included in the canvases.

Post processing:

We used post processing as a feature that might affect the performance of our game and we wanted to test it with the profiler. To use this, we downloaded the **Post Processing Package** of Unity, and applied the effects we wanted, like a blurry background and some slight modifications in lighting and saturation.





Standard Asset Pack:

From the Unity asset store, we downloaded this package- which is very old but some features are still very useful. Our first person player is taken from here. Also we took some particle effects that we modified into weapon shooting effects.

Profile Analyzer:

Profile analyzer is added to Unity editor as a library to evaluate the performance with profiler. It has two abilities as a single view and comparison. So, we can compare our profiler results to increase our performance with the result of it. It helps visualizing frame, thread and marker data. It shows min, max, median, mean and lower/upper quartile results for the selected frame.

Probuilder:

We downloaded the **Probuilder Package** from Unity’s Package Manager. Pro-builder is a tool of easy and quick modeling that can be done right inside unity without using external 3D modeling software. When creating the level design for our game, we wanted to have some walls that are impossible to be passed by only jumping, so that we would force our player to use the gravity switch game mechanic.

Progrids:

We downloaded the **Progrids Package** from Unity's Package Manager. Pro-grids is a package that is mostly used together with probuilder. It allows the user to generate grids at the density and distance they desire. When used together with probuilder we can easily create walls at the distance we want by dragging the corners and snipping.

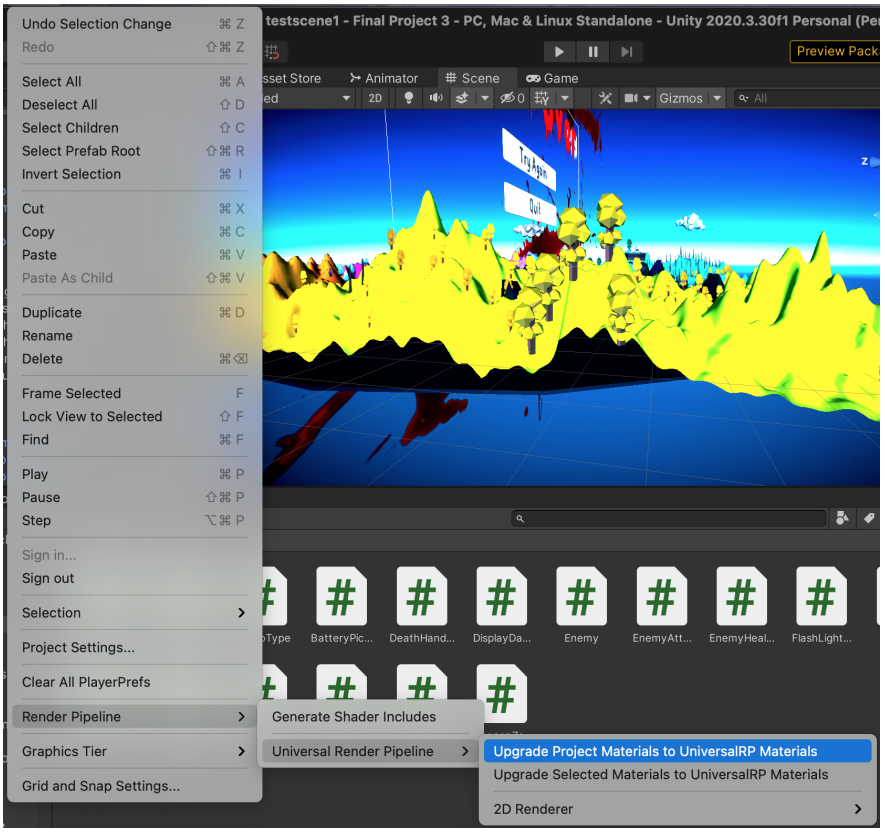
Terrain Tools:

We downloaded the **Terrain Tools Package** from Unity's Package Manager, and used it to define the boundaries of the game world. Using this tool, we have created colorful materials matching our game world and have also added lowpoly trees.

We Used Also

Universal Render Pipeline (URP):

The Universal Render Pipeline (URP) is a prebuilt Scriptable Render Pipeline, made by Unity. We have chosen this render pipeline over the Lightweight Render Pipeline (LWRP), because the asset packages that we were using required URP, and they would appear as (pink) no material objects, if we didn't do so. In order to upgrade all existing materials into URP, we had to do the following procedure: Edit → Render Pipeline → Universal Render Pipeline → Upgrade Project Materials to UniversalRP Materials



Performance - Profiler

We use Unity profiler and Profile Analyzer tools at most for the performance evaluation. They give us general and object detail results. They measure CPU usage, GPU usage, rendering performance, memory usage and physics. To see the script details, the codes below are added to the code blocks at the start and end.

```
UnityEngine.Profiling.Profiler.BeginSample("<text>");
```

```
UnityEngine.Profiling.Profiler.EndSample();
```

Lower number of milliseconds on the profiler means faster games. Garbage collector allocation is the amount of memory that is allocated and needs to be cleaned up in a garbage collection, we generally want it to be 0. We see the timing also and a deep profile that shows us detailed analysis. These all provide us a view to find the part that makes our game slow or decrease the performance.

Profile analyzer is a really useful library that we added to our project because we can have single and compared views there. It is also possible to import data to profile analyzer that is recorded on profiler.

We looked at the parameters below for the results seen on profiler:

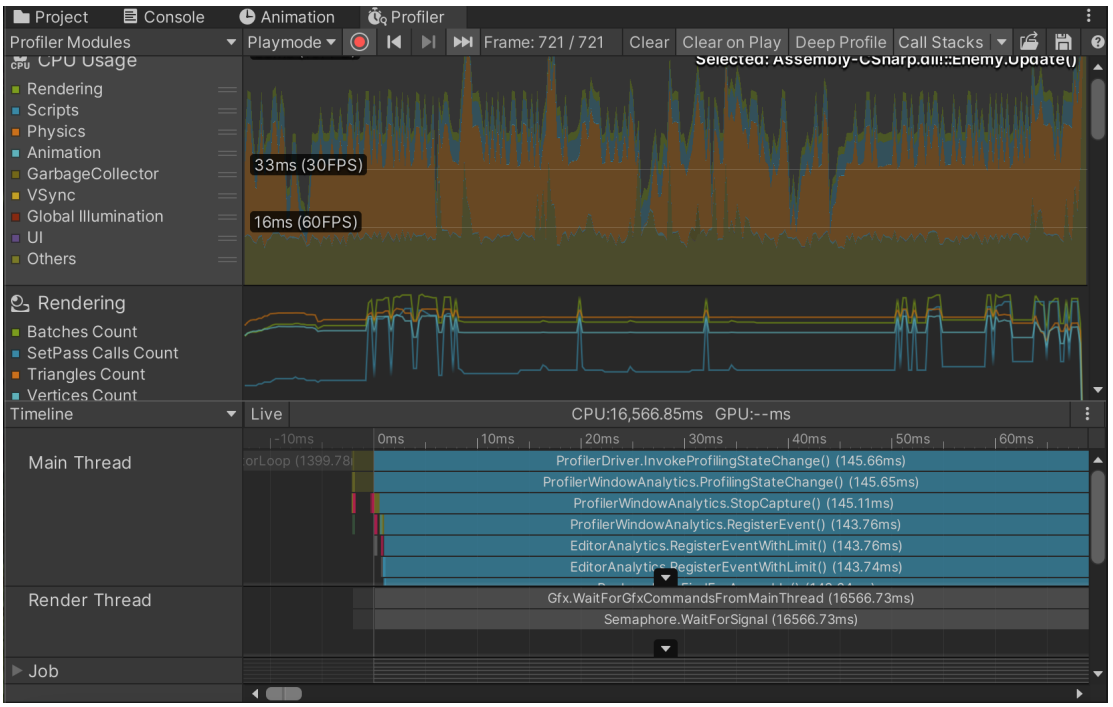
- Open vs Close Terrain Object
- Box Collider vs Mesh Collider for Environment Objects
- Open vs Close Global Volume (Post Process)
- Normal vs Increased Enemy Count
- Stay Ground vs Fly with Gravity and Look Down to Affect Rendering
- Box Collider with Increased Enemy Count

0. Game Scene Default Performance

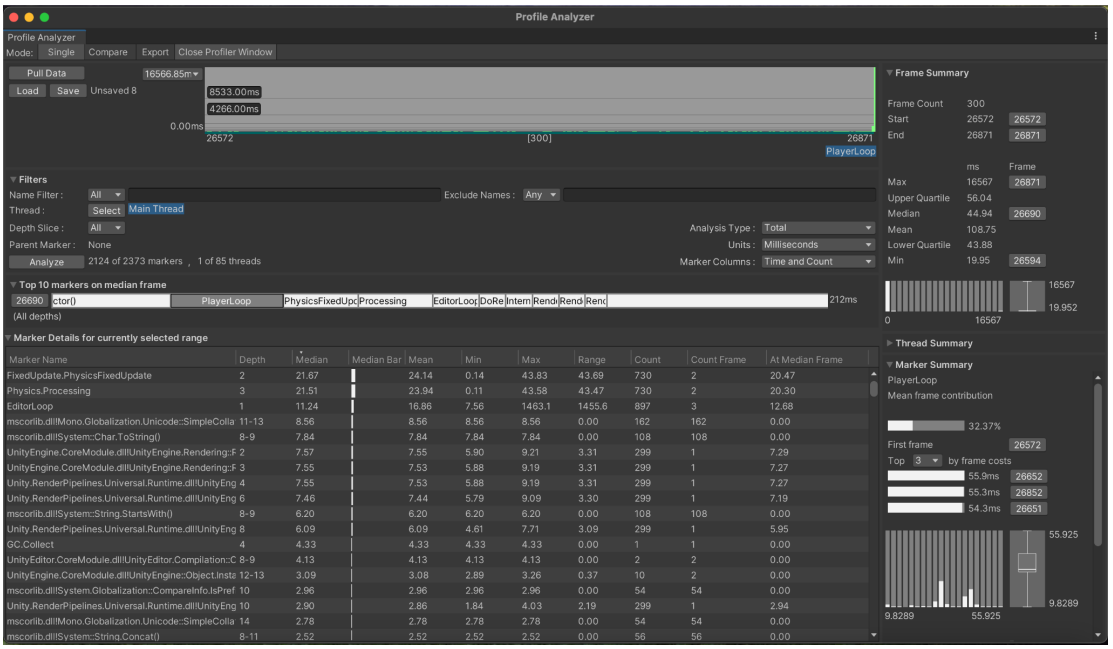
Computer Properties for the Graphs Below:

System Summary	
Operating System	macOS Big Sur (Mid 2014)
CPU / GPU	2,2 GHz Quad-Core Intel Core i7 / Intel Iris Pro 1536 MB
RAM	16 GB 1600 MHz DDR3
Unity Version	2020.3.30f1

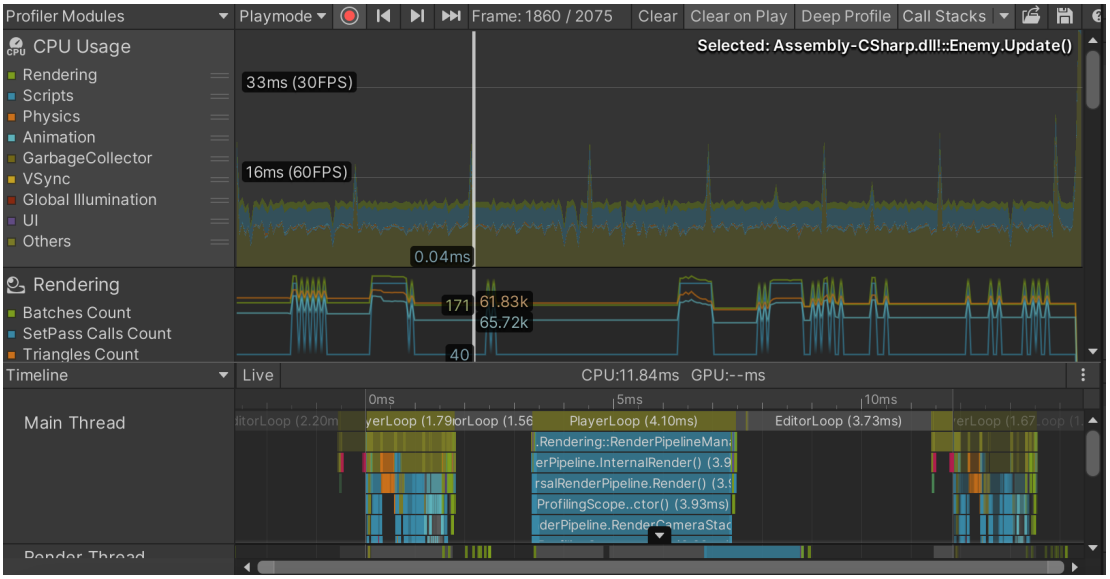
First of all, we see that our scene's gameplay performance is affected by computer performance so much. The graph below shows us high milliseconds with high CPU usage that results with low FPS. Lower performance is also affected by profiler usage. Because our computer properties are not enough, handling both usage of the game and profiler caused performance to be low.



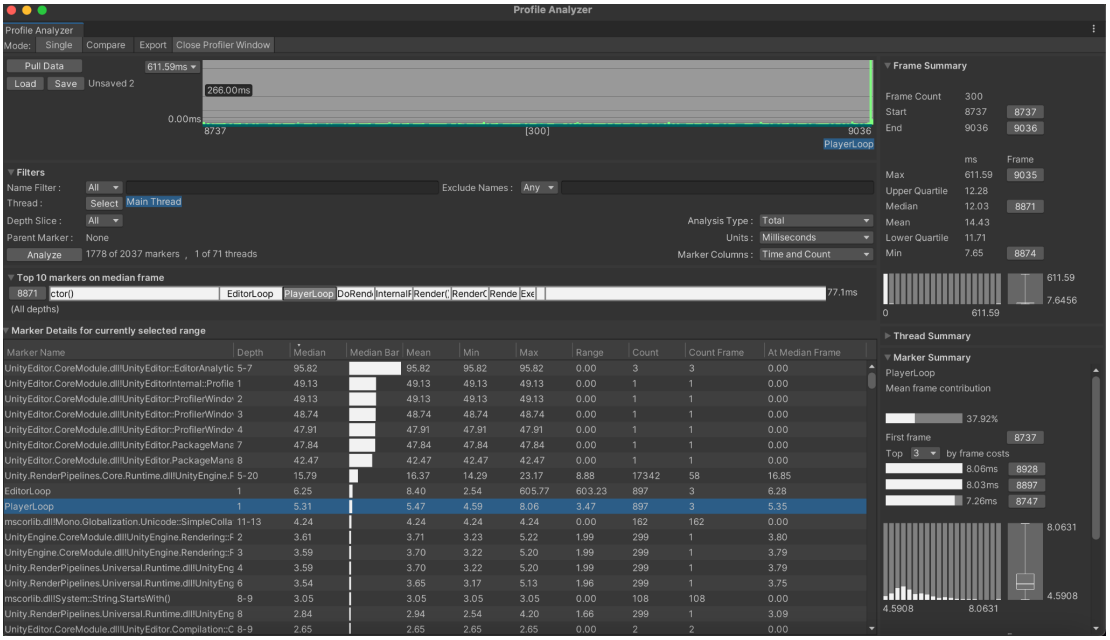
The screenshot below is the details of the profile analyzer. According to the details, physics is the most part of CPU usage.



Other than that, the lowest millisecond result of gameplay is below. It has higher FPS.

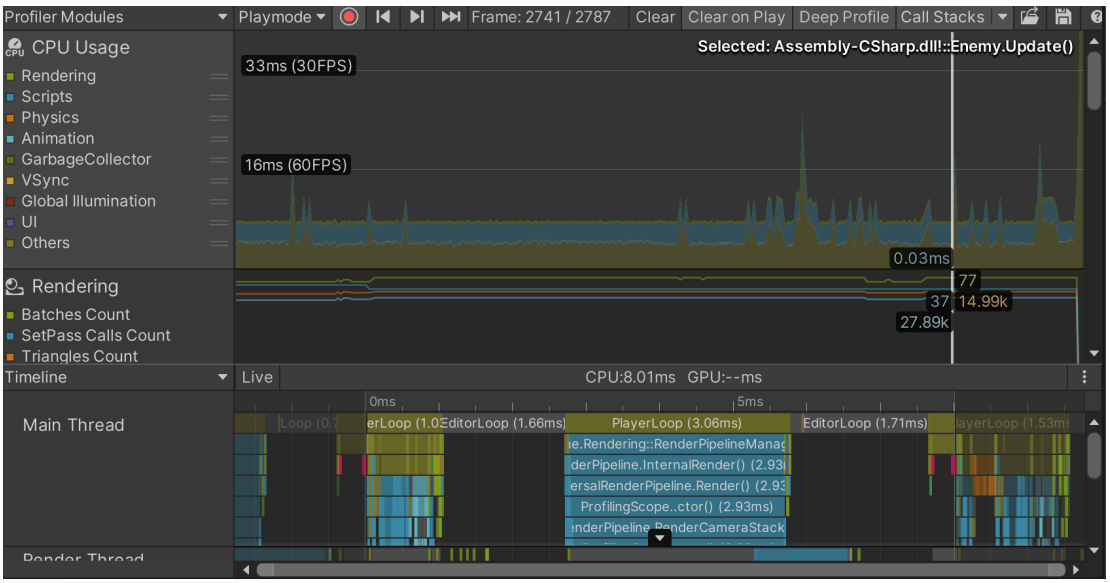


In this time, we see that the most usage for performance is because of Unity Editor and editor loop, player loop are coming after them.

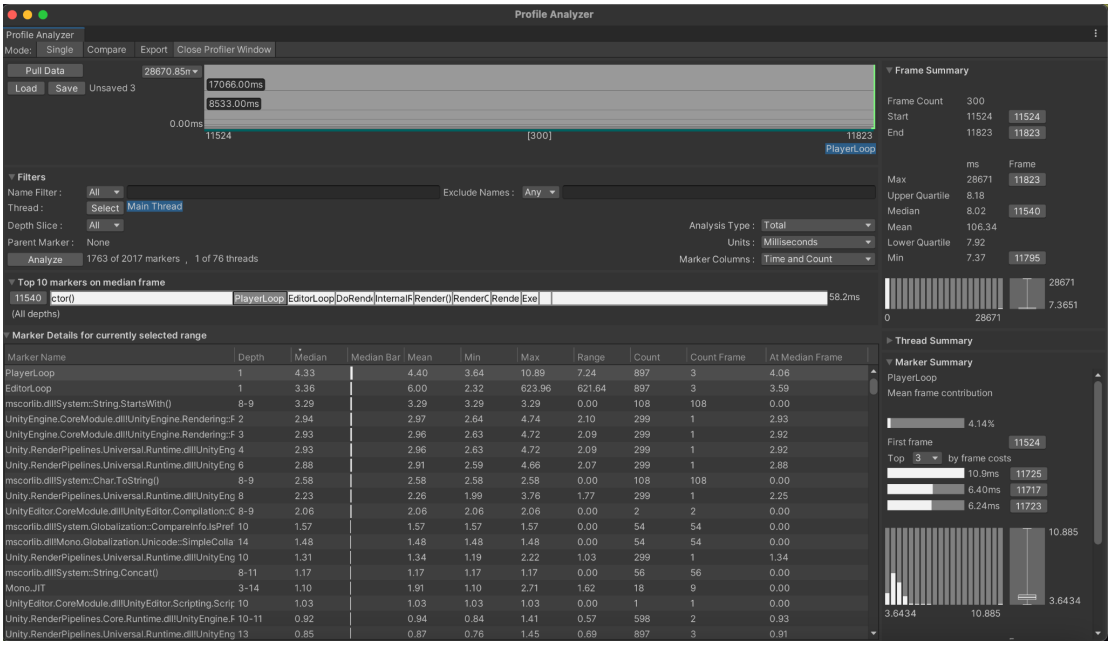


1. Closing Terrain object

To compare normal performance with terrain effect, we closed terrain object because we assumed that it requires much performance. It was our assumption. When we closed the terrain object, it showed higher FPS graphs.

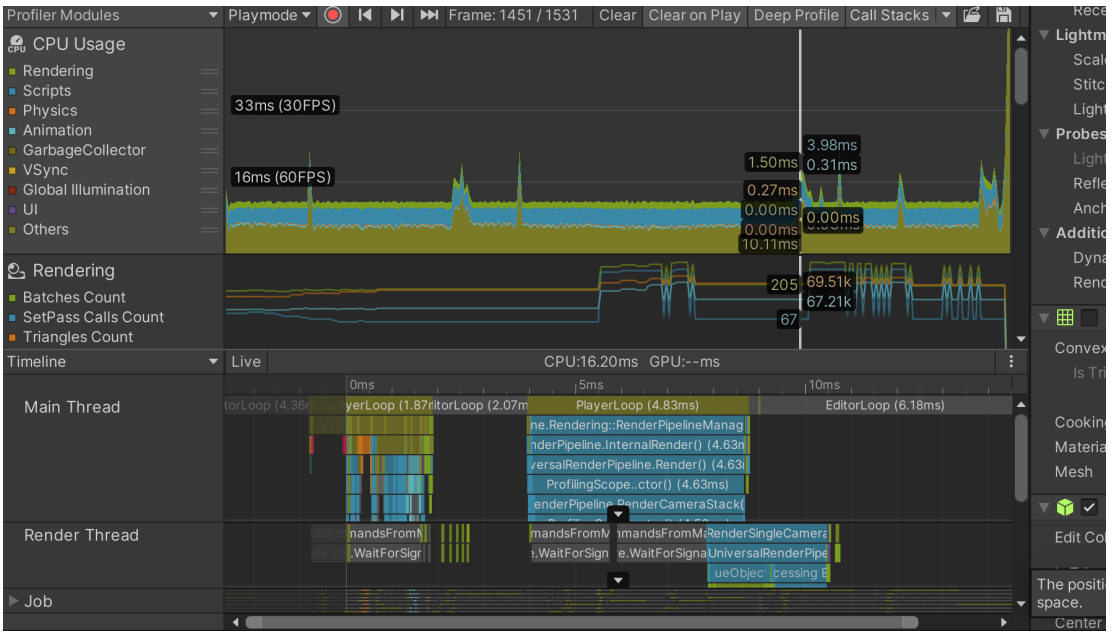


Here, the player loop is the most part of performance usage as it is seen in profile analyzer.

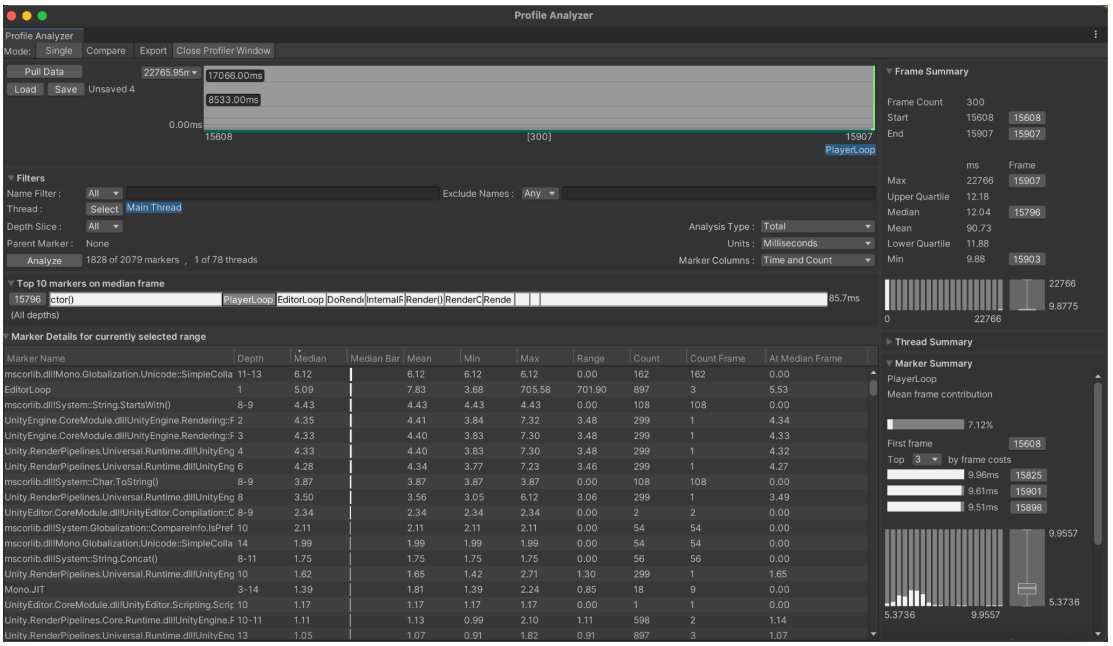


2. Box collider instead of mesh collider for the objects in the environment (not the player or enemies)

Box and mesh collider change did not affect the performance so much. Normally, we expected the milliseconds to be lower when the box collider is used instead of the mesh collider. It did not graph as our assumption. This may be because we do not have so much collision effect in our game. There are environment objects, but the player does not collide with them frequently. Our enemies are working with AI in the background.

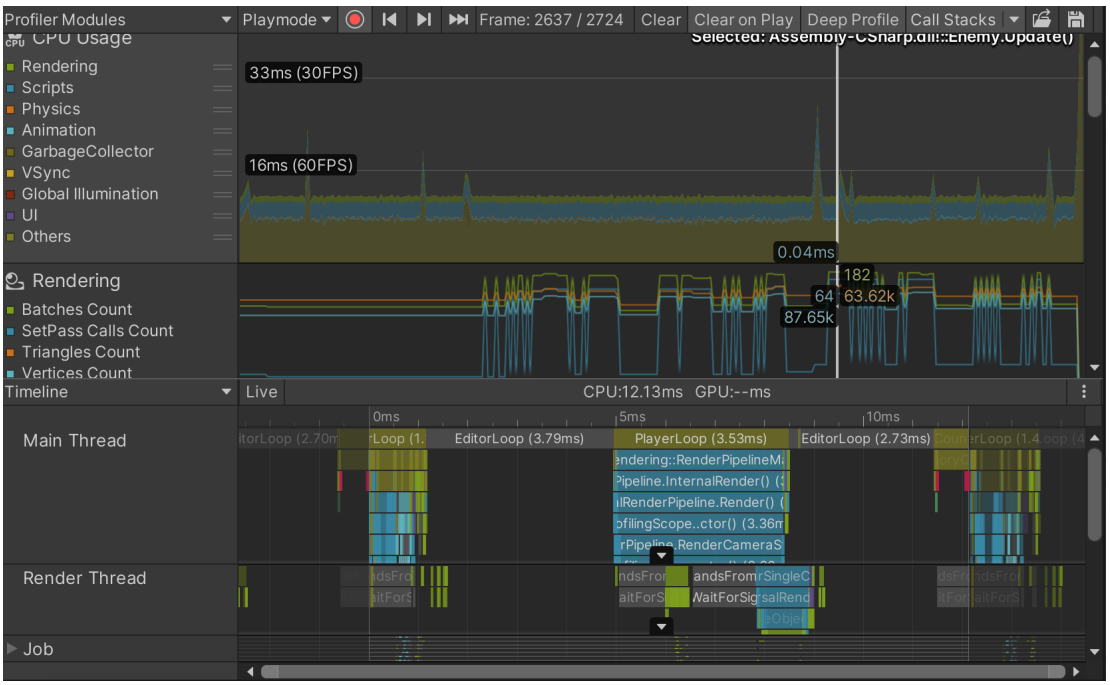


The results of the profile analyzer is again below and it has the same results as above.

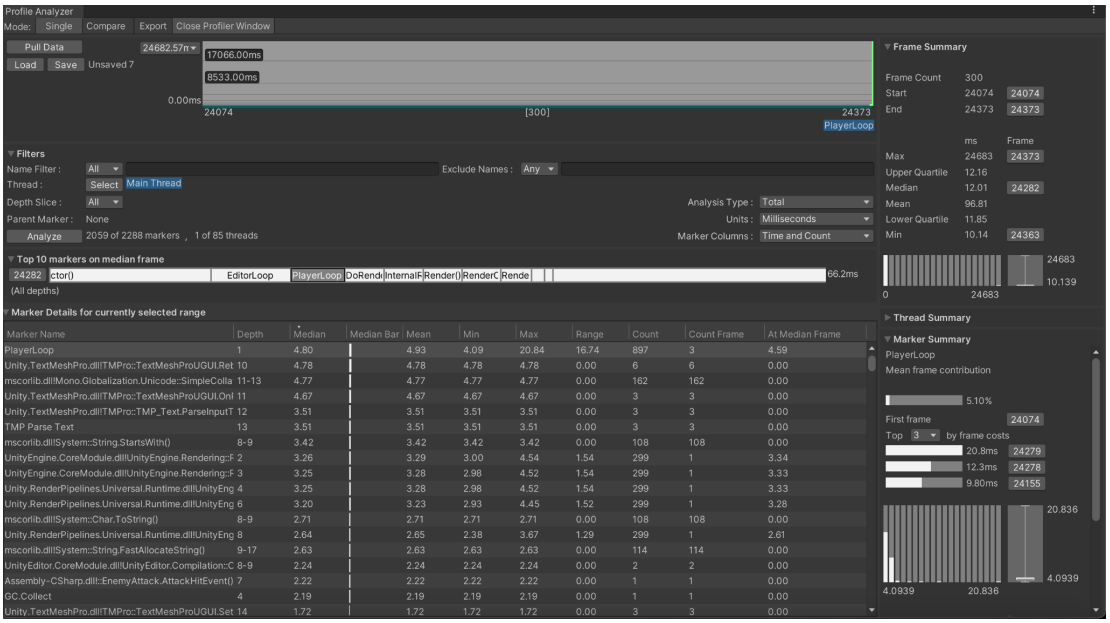


3. Closing global volume (post process) object

When we compared this change with default results of our scene, it shows us that post processing does not affect the performance so much. We expected it to result in lower milliseconds, but it did not change. This might be because we did not use so many features of post processing. Color adjustments might have affected the performance more.

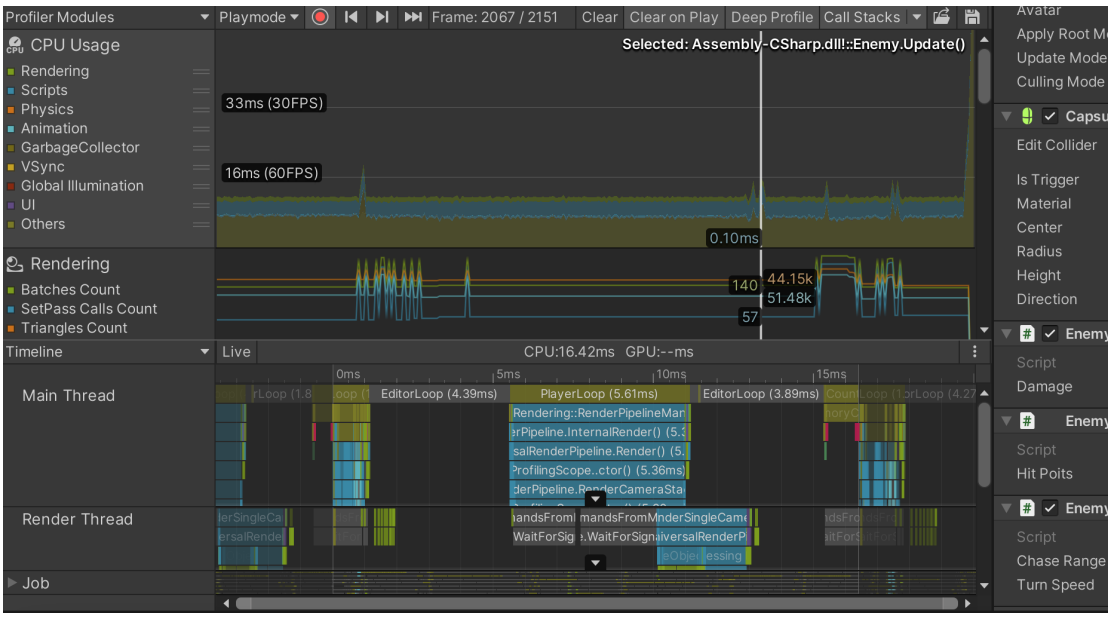


The details of the profile analyzer is below. It shows the most performance usage is by player loop.

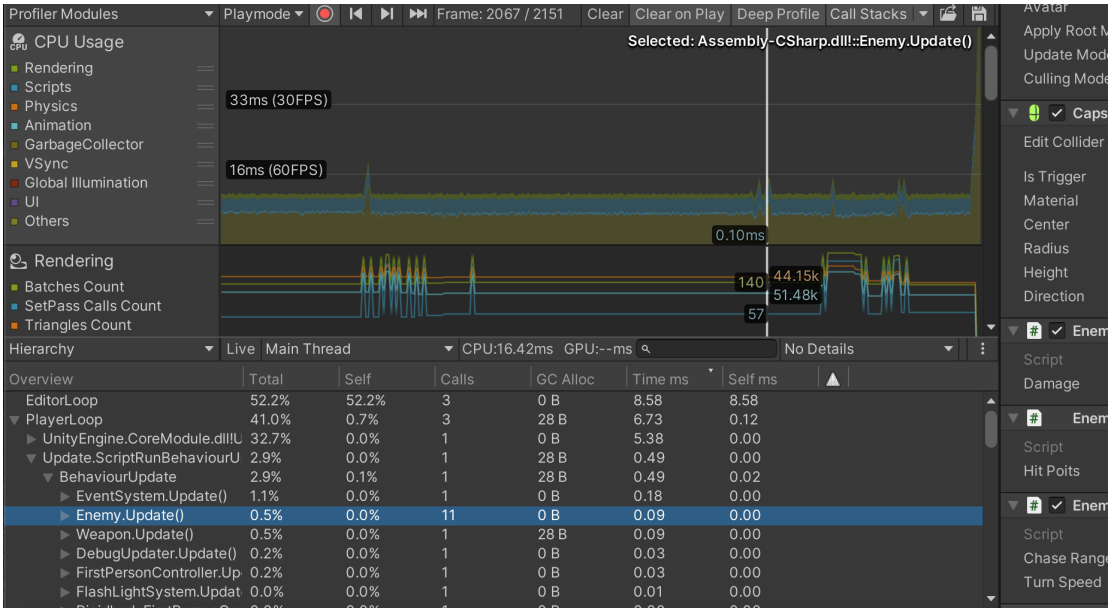


4. Increasing enemy count

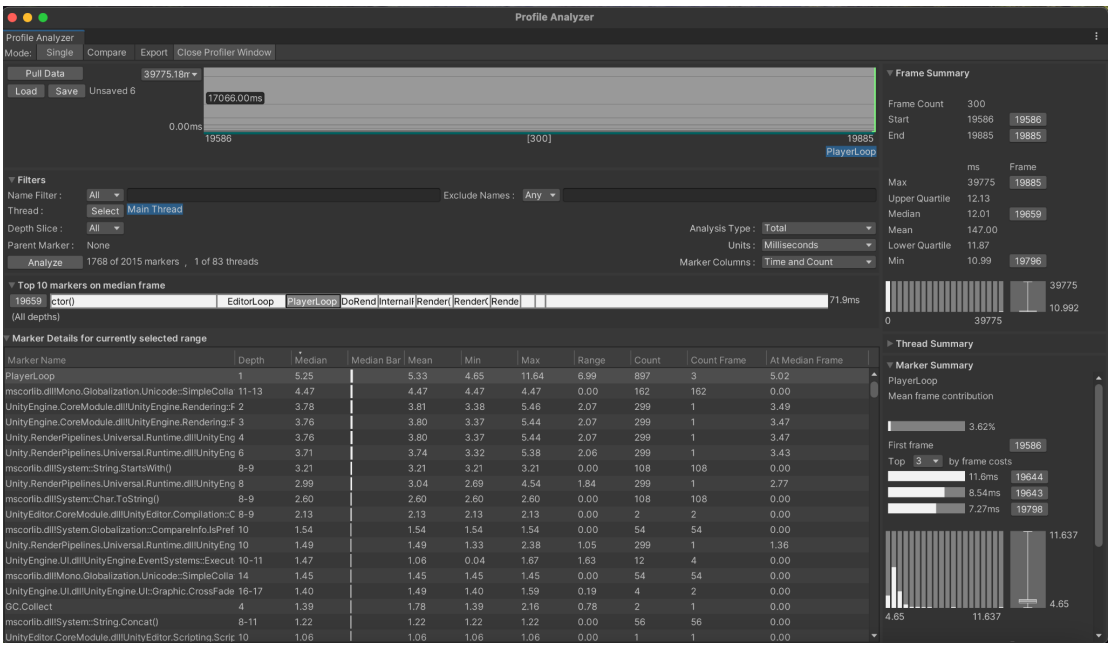
Increased enemy count decreases the performance just a little. We multiplied the enemy count by 2 for our test case. The graphs below show that the FPS result does not change so much, but calling enemy script increases which decrease the performance just by a small amount.



Below we see the hierarchy details with the enemy script because we added profiler code to our enemy script.

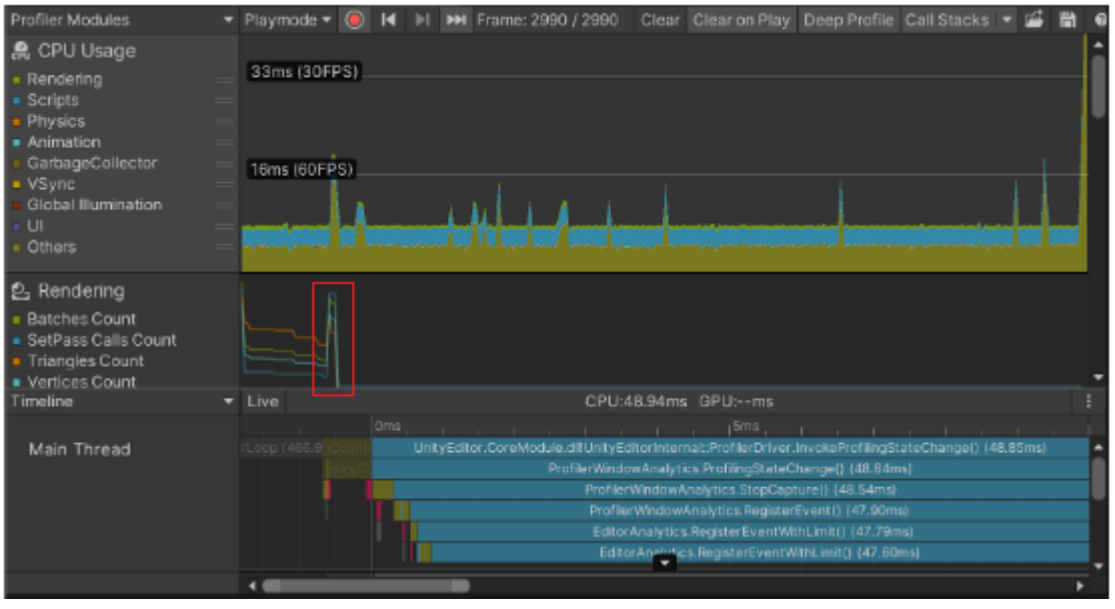


Again profile analyzer results are below. The player loop is the most performance using criteria.

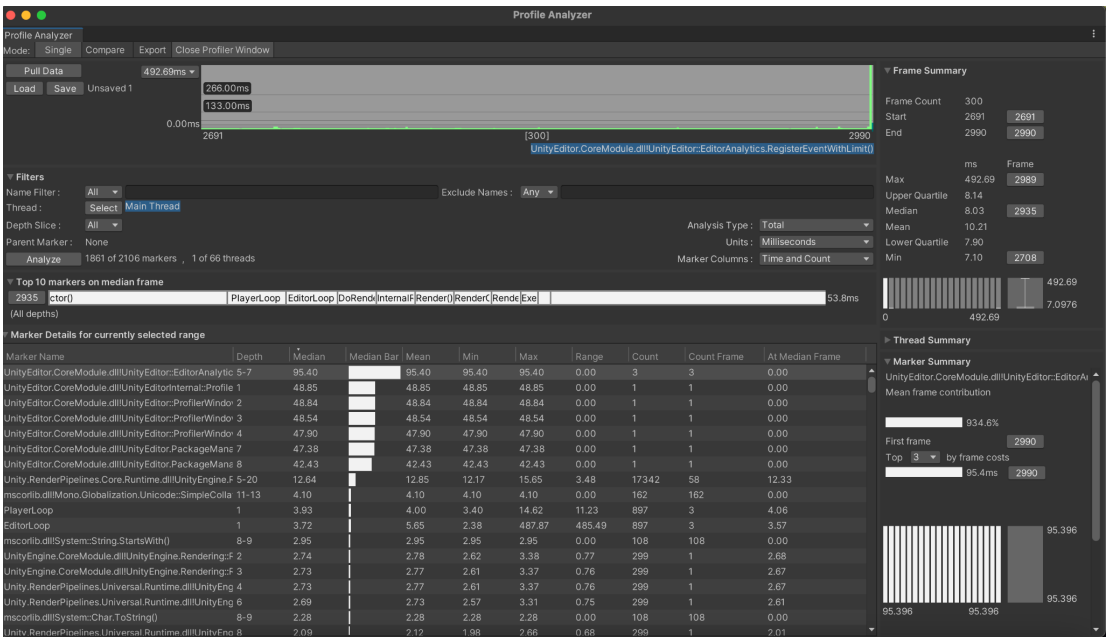


5. Flying with gravity and looking down which affects rendering

Flying with gravity does not affect performance so much, but it affects rendering. When flying is started and the player looks down, rendering increases about 2 times because the player sees so many objects from the top view. The area we see in the red box is the result of this. On the other hand, below it is seen the rendering results are lost when flying is continued to a much higher place because we do not see any object any more.

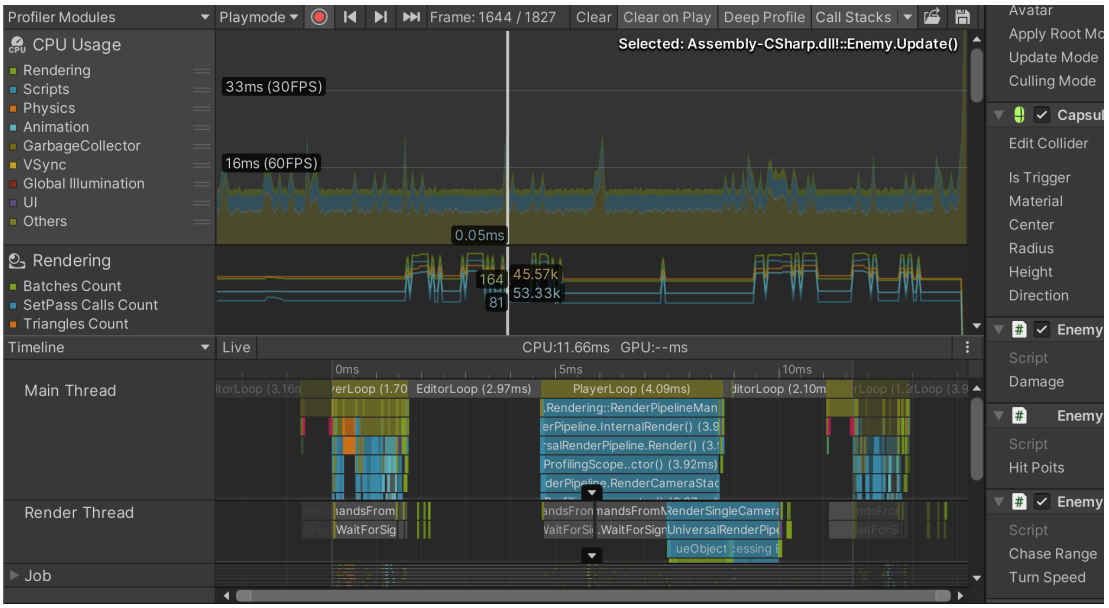


Profile analyzer details are given below.

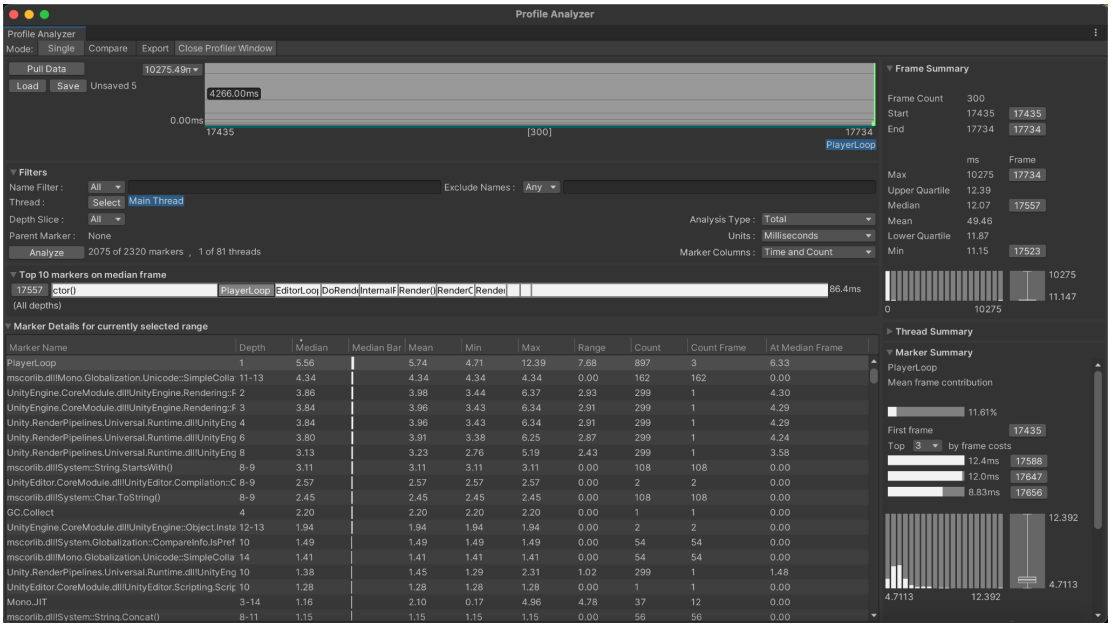


6. Box Collider with increased enemy count

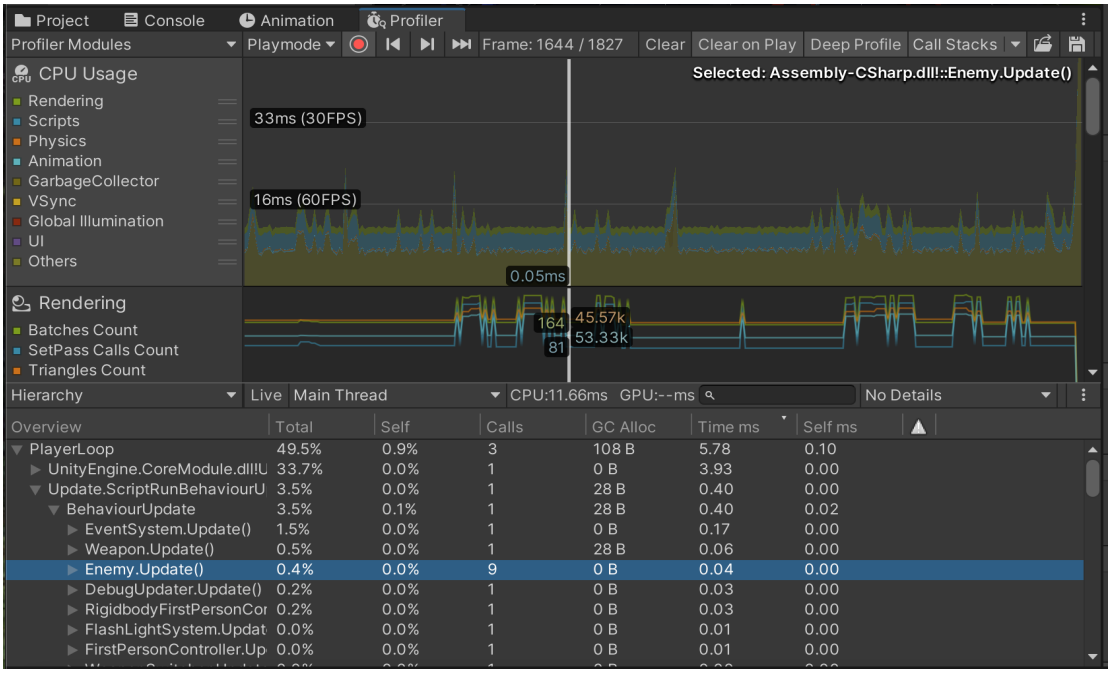
As explained above in the box vs mesh collider section, box collider does not affect the performance so much because we do not have so many collisions. However, when we combine it with increased enemy count, they decrease the performance a little because enemy script calls are also increased.



The profile analyzer details are given below. The most usage is done by player loop.



As seen below, the enemy script call is high when compared with default play. We see this result thanks to the code we added to the enemy script. We added the weapon script too, but did not use it for performance evaluation.



What To Do Next

We planned our game and designed accordingly, but there are still areas to be improved. For example, even if we collect batteries and increase lightning, the difference is not seen very well. We can change the screen colors to show this or we can add other collectable objects. We thought to have 4 different element stones that can be collected and if all are completed, the game ends as a win result. However, we did not add a win condition for now which is an improvement area for our game.