Course Project

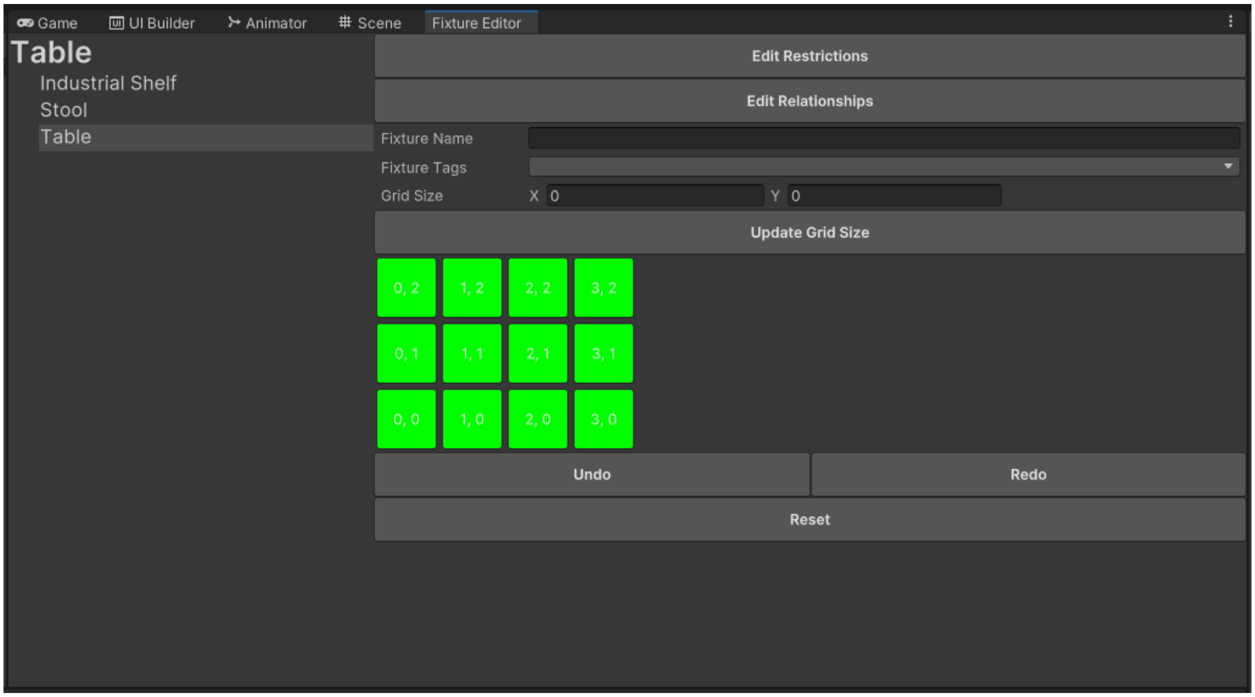
Video Report

Here is the link to the video report,
https://drive.google.com/drive/folders/1zyZWXFRNtDis4ngX_xzehD3haltFgClw?usp=sharing

Additional Assignment 1 Deliverables:
https://docs.google.com/document/d/1ReWeXFyYb3p7qzbSJtet8tDHzmTDXRApR1e44k2dDOk/edit?usp=sharing

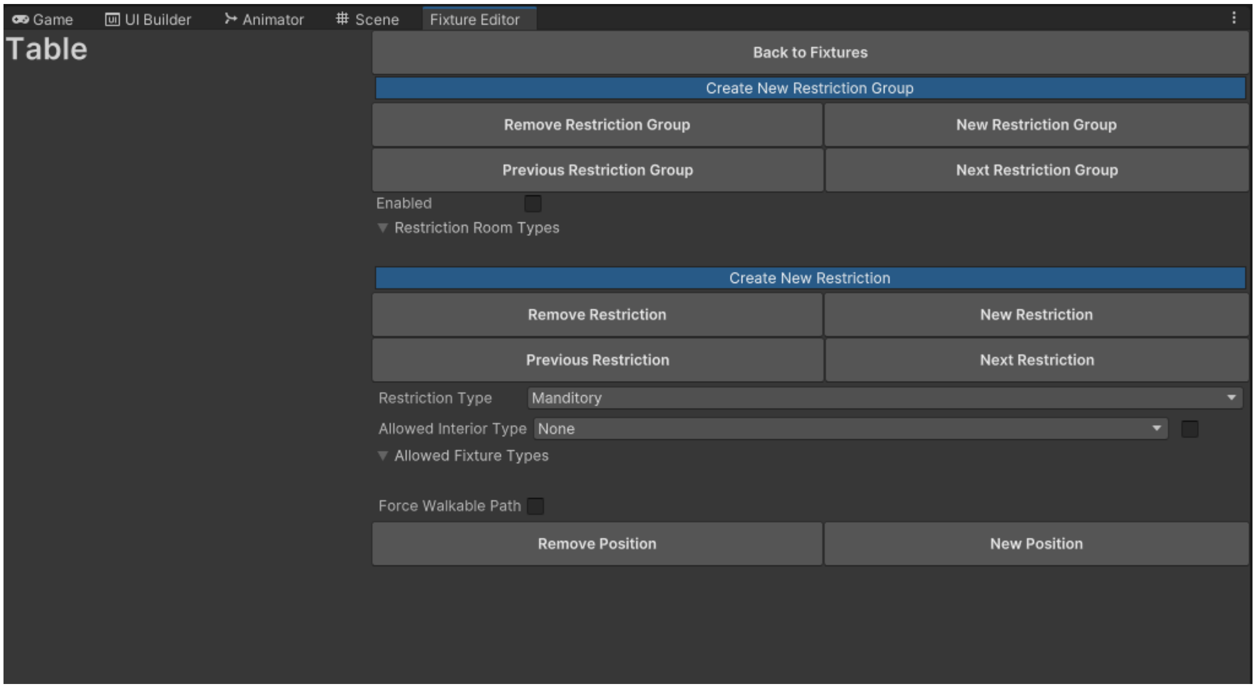
Game Design Pattern Improvements from the Previous Assignment **(20/100)**

Denzil



Fixture Name
Fixture Tags
Grid Size X 0 Y 0

0, 2	1, 2	2, 2	3, 2
0, 1	1, 1	2, 1	3, 1
0, 0	1, 0	2, 0	3, 0



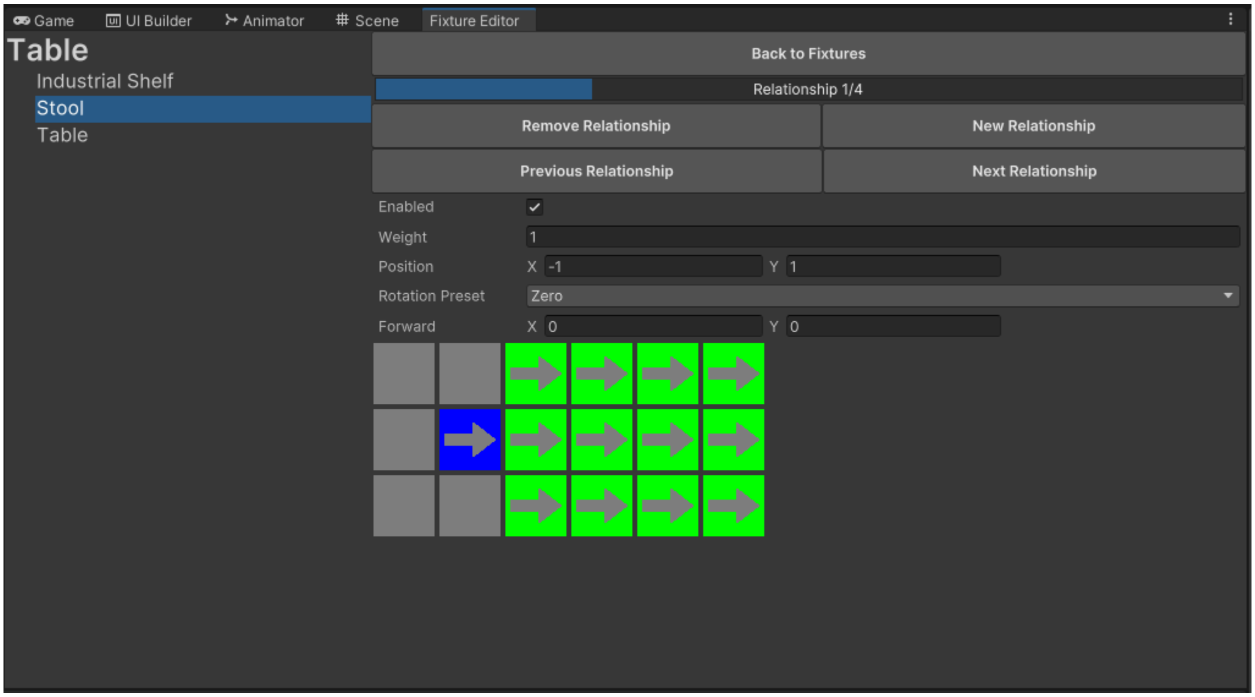
Table

Enabled □

Restriction Type: Manditory

Allowed Interior Type: None □

Force Walkable Path □



Table

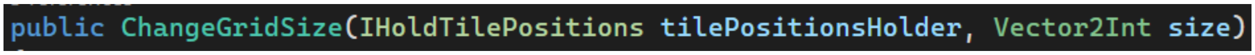
Enabled ☑

Weight: 1

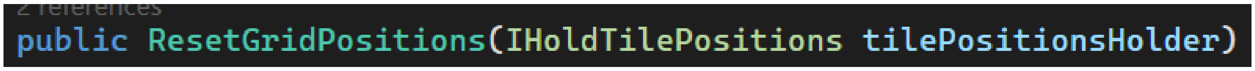
Position: X -1 Y 1

Rotation Preset: Zero

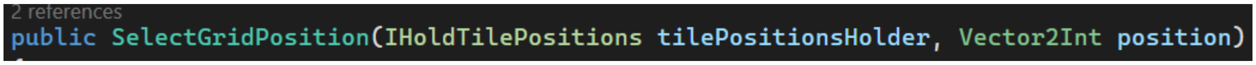
Forward: X 0 Y 0



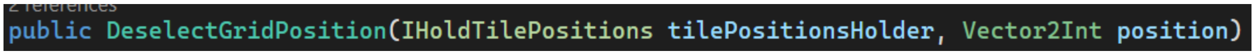
```
public ChangeGridSize(IHoldTilePositions tilePositionsHolder, Vector2Int size)
```



```
2 references
public ResetGridPositions(IHoldTilePositions tilePositionsHolder)
```



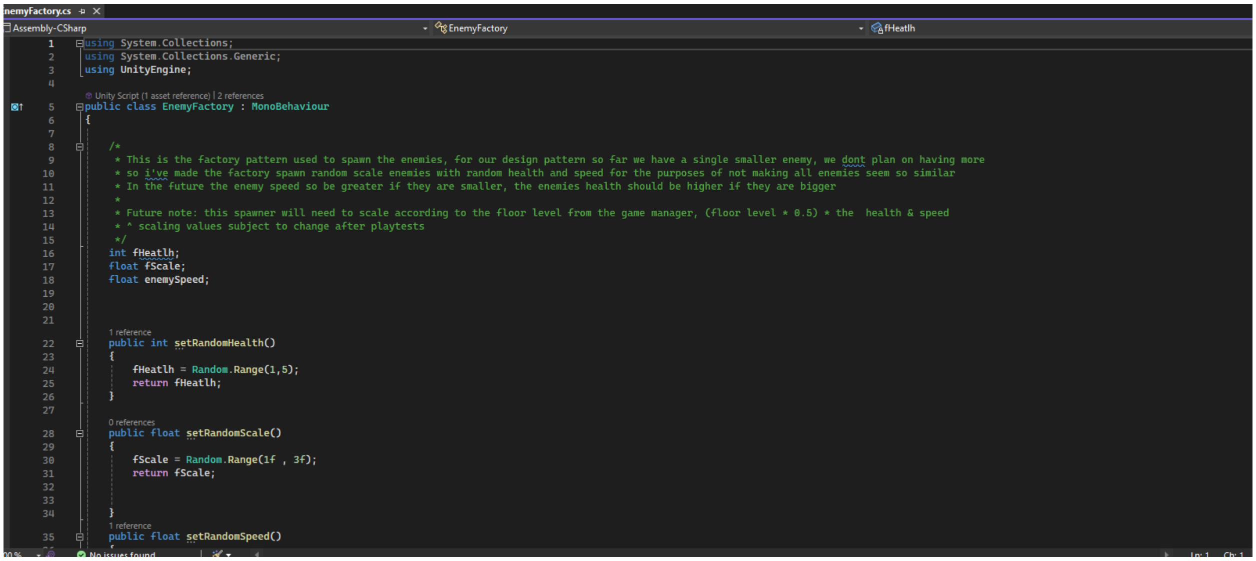
```
2 references
public SelectGridPosition(IHoldTilePositions tilePositionsHolder, Vector2Int position)
```



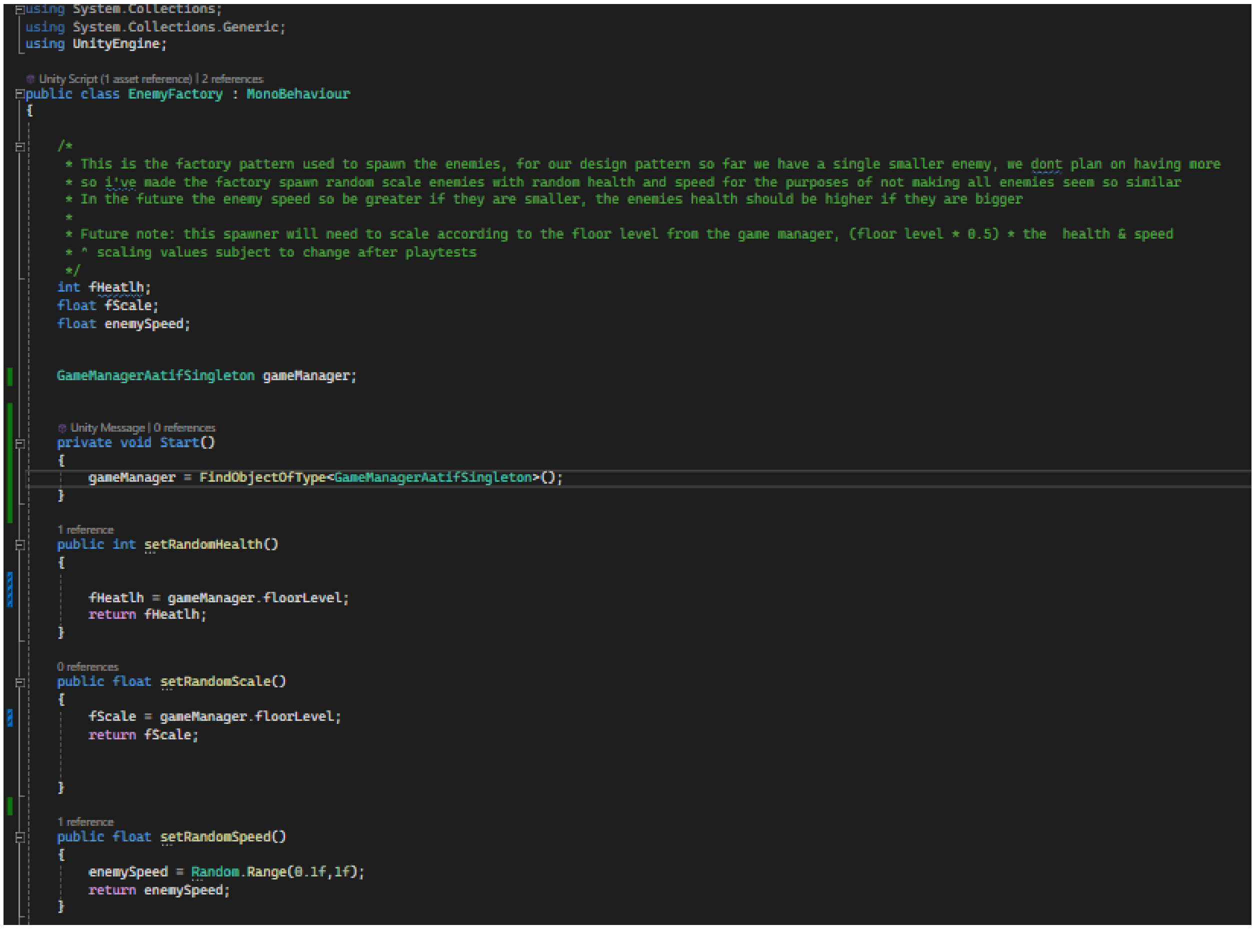
```
2 references
public DeselectGridPosition(IHoldTilePositions tilePositionsHolder, Vector2Int position)
```

My improvements from the previous assignment come in the form of a new editor tool, and class adjustments for the command design pattern implemented. The editor window took me around 2 weeks to complete, so most of my time was taken up by it. The part related to the command design pattern implementation is showcased in the top picture, but I also included some other pictures to show off what I did in the editor. This editor window allows me to easily modify the fixtures that are placed through the procedural generation system. This is absolutely necessary to achieve good looking procedural generation, as it will need many tweaks, restrictions, and relationships to appear realistic. I also created a new IHoldTilePositions interface which the various tile changing commands use when changing the tile positions of the object they are called on. This allows me to implement the tile changing commands with more than just the RoomData class as it was before.

Aatif



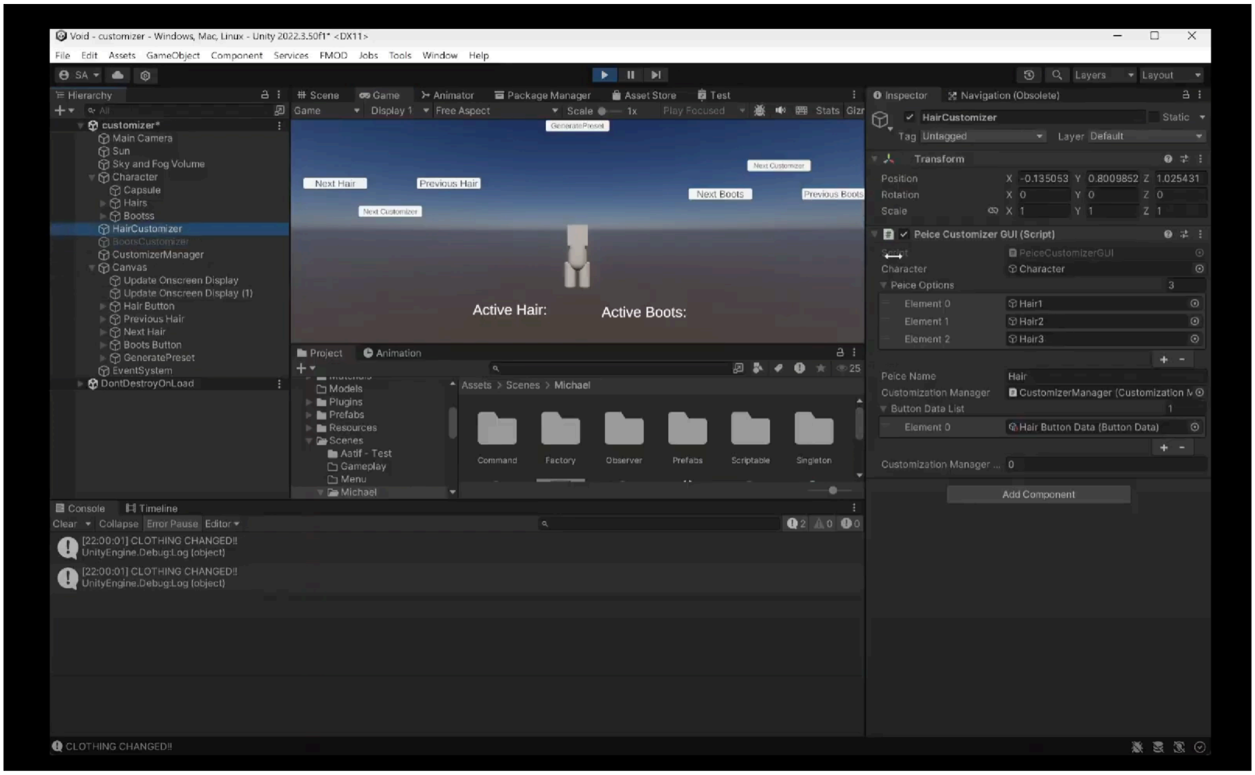
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyFactory : MonoBehaviour
{

    /*
     * This is the factory pattern used to spawn the enemies, for our design pattern so far we have a single smaller enemy, we dont plan on having more
     * so i've made the factory spawn random scale enemies with random health and speed for the purposes of not making all enemies seem so similar
     * In the future the enemy speed so be greater if they are smaller, the enemies health should be higher if they are bigger
     *
     * Future note: this spawner will need to scale according to the floor level from the game manager, (floor level * 0.5) * the  health & speed
     * ^ scaling values subject to change after playtests
     */
    int fHeatlh;
    float fScale;
    float enemySpeed;



    public int setRandomHealth()
    {
        fHeatlh = Random.Range(1,5);
        return fHeatlh;
    }

    public float setRandomScale()
    {
        fScale = Random.Range(1f , 3f);
        return fScale;


    }
    public float setRandomSpeed()
```



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script (1 asset reference) | 2 references
public class EnemyFactory : MonoBehaviour
{

    /*
     * This is the factory pattern used to spawn the enemies, for our design pattern so far we have a single smaller enemy, we dont plan on having more
     * so i've made the factory spawn random scale enemies with random health and speed for the purposes of not making all enemies seem so similar
     * In the future the enemy speed so be greater if they are smaller, the enemies health should be higher if they are bigger
     *
     * Future note: this spawner will need to scale according to the floor level from the game manager, (floor level * 0.5) * the  health & speed
     * ^ scaling values subject to change after playtests
     */
    int fHeatlh;
    float fScale;
    float enemySpeed;


    GameManagerAatifSingleton gameManager;


    Unity Message | 0 references
    private void Start()
    {
        gameManager = FindObjectOfType<GameManagerAatifSingleton>();
    }

    1 reference
    public int setRandomHealth()
    {

        fHeatlh = gameManager.floorLevel;
        return fHeatlh;
    }

    0 references
    public float setRandomScale()
    {
        fScale = gameManager.floorLevel;
        return fScale;


    }

    1 reference
    public float setRandomSpeed()
    {
        enemySpeed = Random.Range(0.1f,1f);
        return enemySpeed;
    }
```

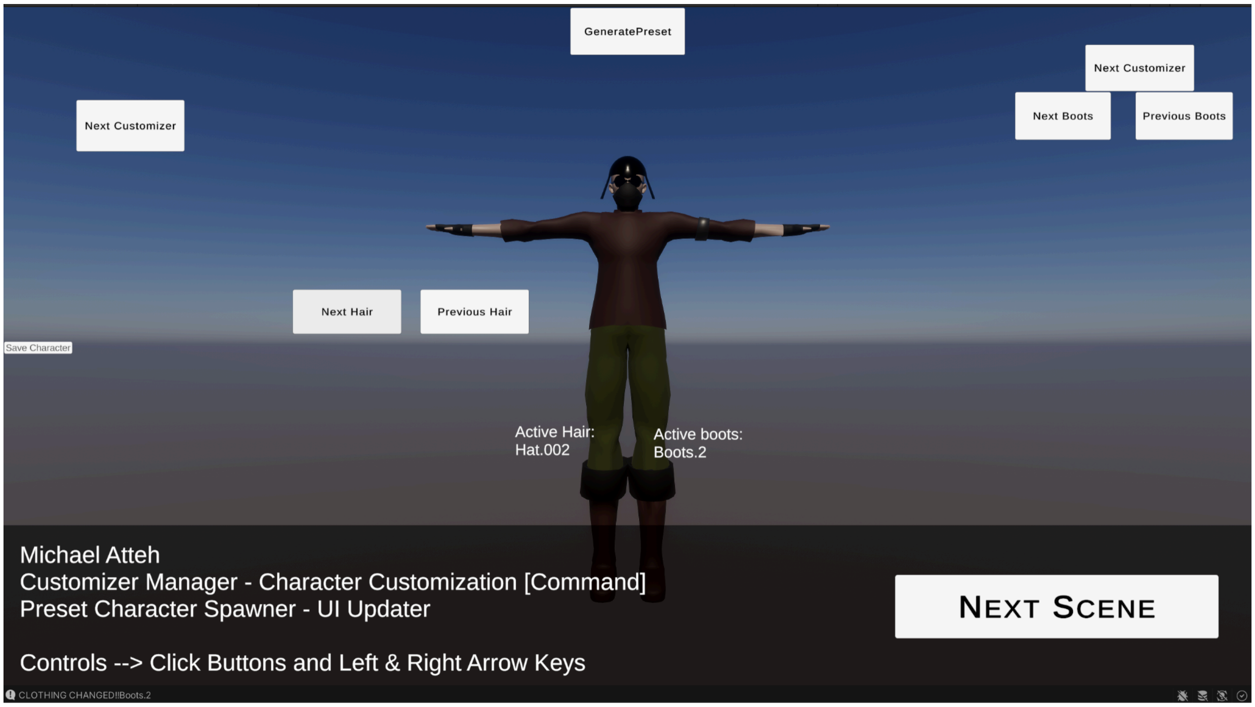
The first image is the factory pattern in the prior assignment, the second image is the factory pattern in this assignment. The way I made the design pattern better was to combine two of them. The enemy factory gets a reference to the game manager which is a singleton because there only needs to be 1 game manager. With this reference it then takes the floor level into consideration when spawning enemies. This is better than before because the game can now scale more evenly as the levels go on, this is good because without the merging of these two scripts there would need to be a new enemy factory every in game level to control the enemy that spawns. This is good for the game because if the enemies on floor 5 felt like the enemies on floor 1 the game would be very boring, using this method to scale enemies is helpful because there will always be a game manager to feed info to the factory to instruct it with the difficulty of enemies that need to be spawn.

Michael



HairCustomizer
Tag Untagged Layer Default
Transform
Position X -0.135053 Y 0.8009852 Z 1.025431
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Peice Customizer GUI (Script)
Script PeiceCustomizerGUI
Character Character
Peice Options 3
Element 0 Hair1
Element 1 Hair2
Element 2 Hair3
Peice Name Hair
Customization Manager CustomizerManager (Customization M
Button Data List 1
Element 0 Hair Button Data (Button Data)
Customization Manager ... 0

```
[22:00:01] CLOTHING CHANGED!!
UnityEngine.Debug:Log (object)
[22:00:01] CLOTHING CHANGED!!
UnityEngine.Debug:Log (object)
CLOTHING CHANGED!!
```



Active Hair:
Hat.002

Active boots:
Boots.2

Michael Atteh
Customizer Manager - Character Customization [Command]
Preset Character Spawner - UI Updater

Controls --> Click Buttons and Left & Right Arrow Keys

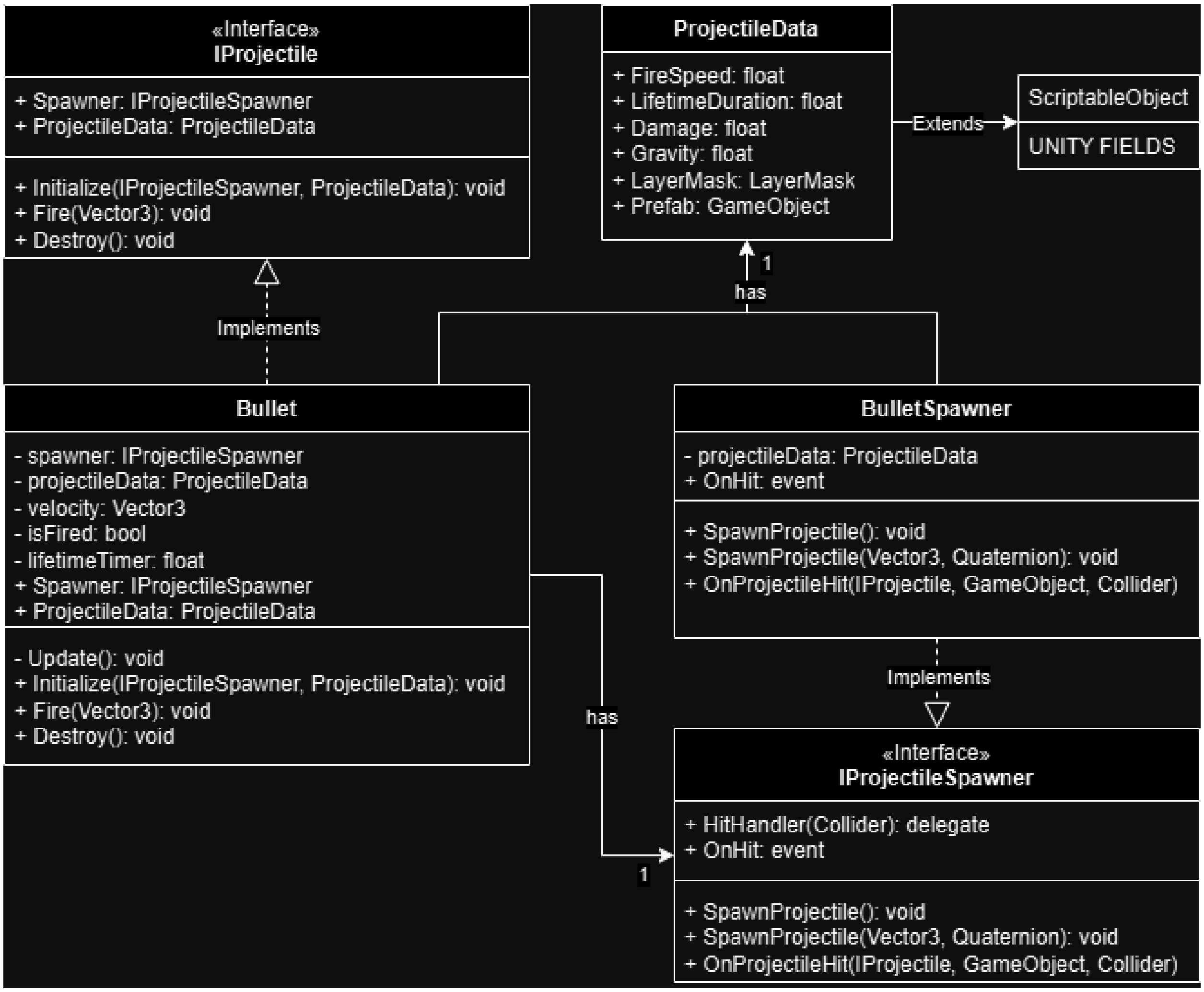
```
CLOTHING CHANGED!!Boots.2
```

My improvements made were the introduction of a fully modelled character into the game, with customizations to use for both the hat and the boots. This helps test the interactive scene in action and improve the experience for the player's by giving them an actual character to see and choose customizations from. The second improvement I made is the addition of the ui indication of the objects using the observer pattern. I implemented live updating of the current item active as a customization option so the user can easily identify what object has been selected when making their customizations.

Optimization Design Patterns **(30/100)**

Flyweight **[Denzil]**

Diagram

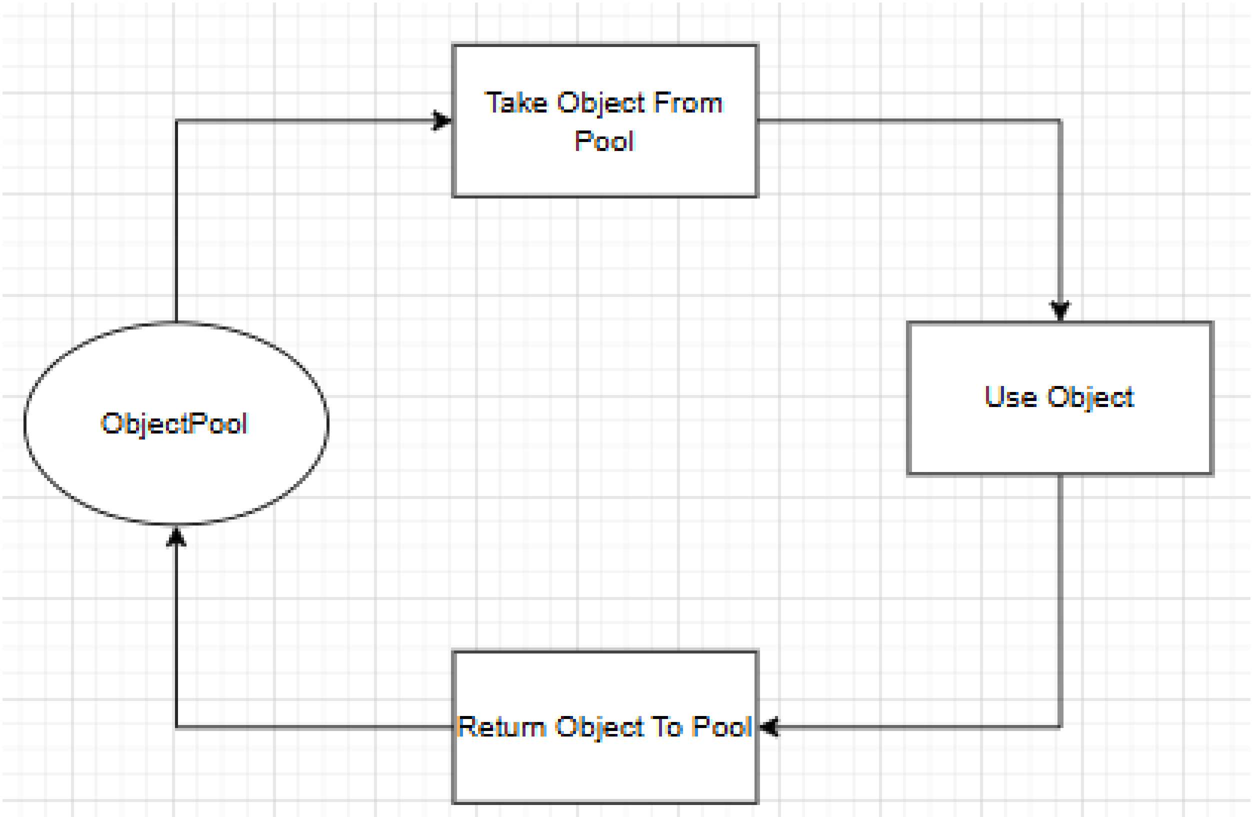

«Interface»
IProjectile
+ Spawner: IProjectileSpawner
+ ProjectileData: ProjectileData
+ Initialize(IProjectileSpawner, ProjectileData): void
+ Fire(Vector3): void
+ Destroy(): void
ProjectileData
+ FireSpeed: float
+ LifetimeDuration: float
+ Damage: float
+ Gravity: float
+ LayerMask: LayerMask
+ Prefab: GameObject
Extends
ScriptableObject
UNITY FIELDS
1
has
Implements
Bullet
- spawner: IProjectileSpawner
- projectileData: ProjectileData
- velocity: Vector3
- isFired: bool
- lifetimeTimer: float
+ Spawner: IProjectileSpawner
+ ProjectileData: ProjectileData
- Update(): void
+ Initialize(IProjectileSpawner, ProjectileData): void
+ Fire(Vector3): void
+ Destroy(): void
BulletSpawner
- projectileData: ProjectileData
+ OnHit: event
+ SpawnProjectile(): void
+ SpawnProjectile(Vector3, Quaternion): void
+ OnProjectileHit(IProjectile, GameObject, Collider)
Implements
has
1
«Interface»
IProjectileSpawner
+ HitHandler(Collider): delegate
+ OnHit: event
+ SpawnProjectile(): void
+ SpawnProjectile(Vector3, Quaternion): void
+ OnProjectileHit(IProjectile, GameObject, Collider)

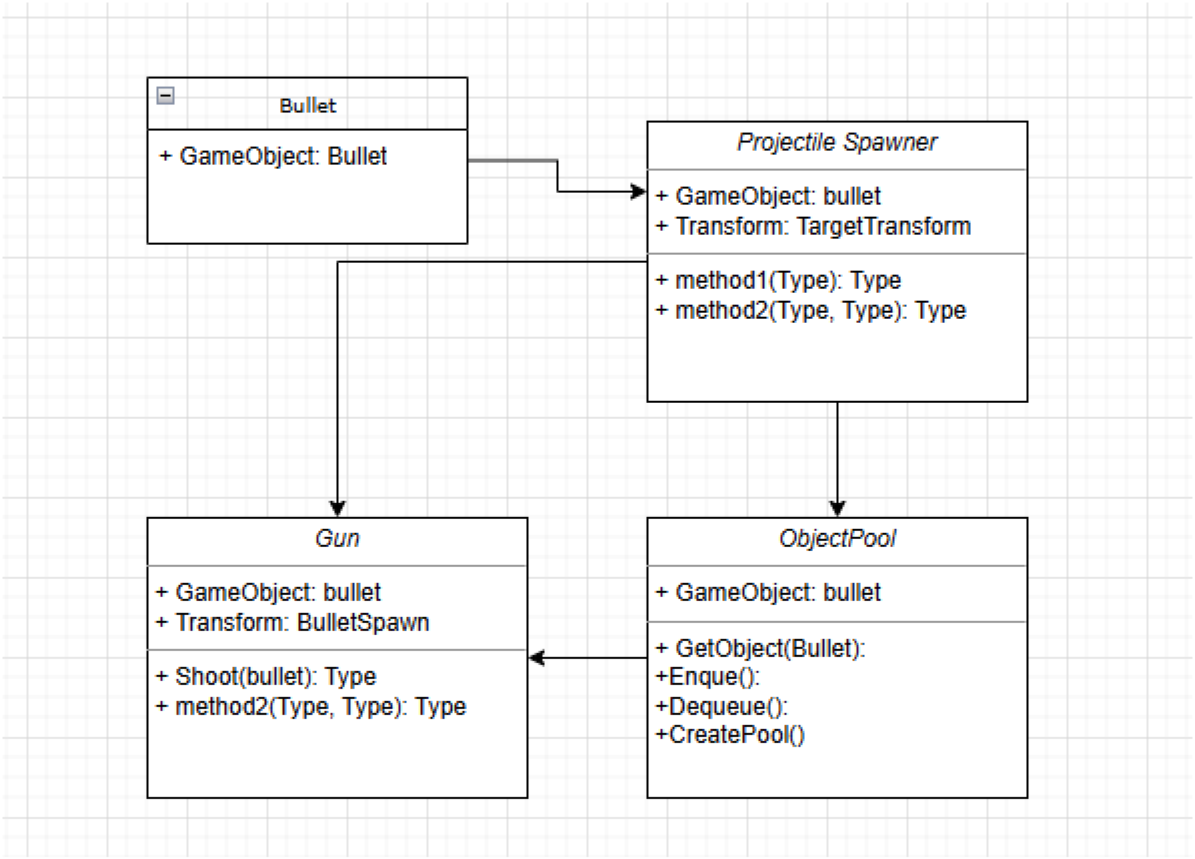
Explanation

To implement the flyweight optimization design pattern, I first had to find a place where using it would be meaningful. My group decided to do a first person shooter for our GDW project, so our game will have a fair amount of bullets/projectiles. Since common projectiles will store common data, it makes sense to use the flyweight pattern to save memory. I don't want to look myself out of expanding the projectiles, so I can only define base variables shared by all projectiles. As shown in the UML diagram above, those variables include the FireSpeed, LifetimeDuration, Damage, Gravity, LayerMask, and Prefab variables. By separating out the variables into a separate class instance that all projectiles can store a reference to, I reduce the memory costs of each individual projectile. While with this current version, the saved memory may not be much, if I were to add a homing projectile that bounces and pieces objects, I would need to define more variables, and in turn take up more memory. Rather than making the projectile memory cost bigger and bigger, I can cap it to the cost of a pointer for most of the information it will need. Since my group's game is multiplayer, it's essential to keep the performance running well, since a lot of power will go into synchronizing client states. If 4 players are all firing fast firing guns, that memory can take a hit on the overall performance.

Object Pool **[Aatif]**

Diagram



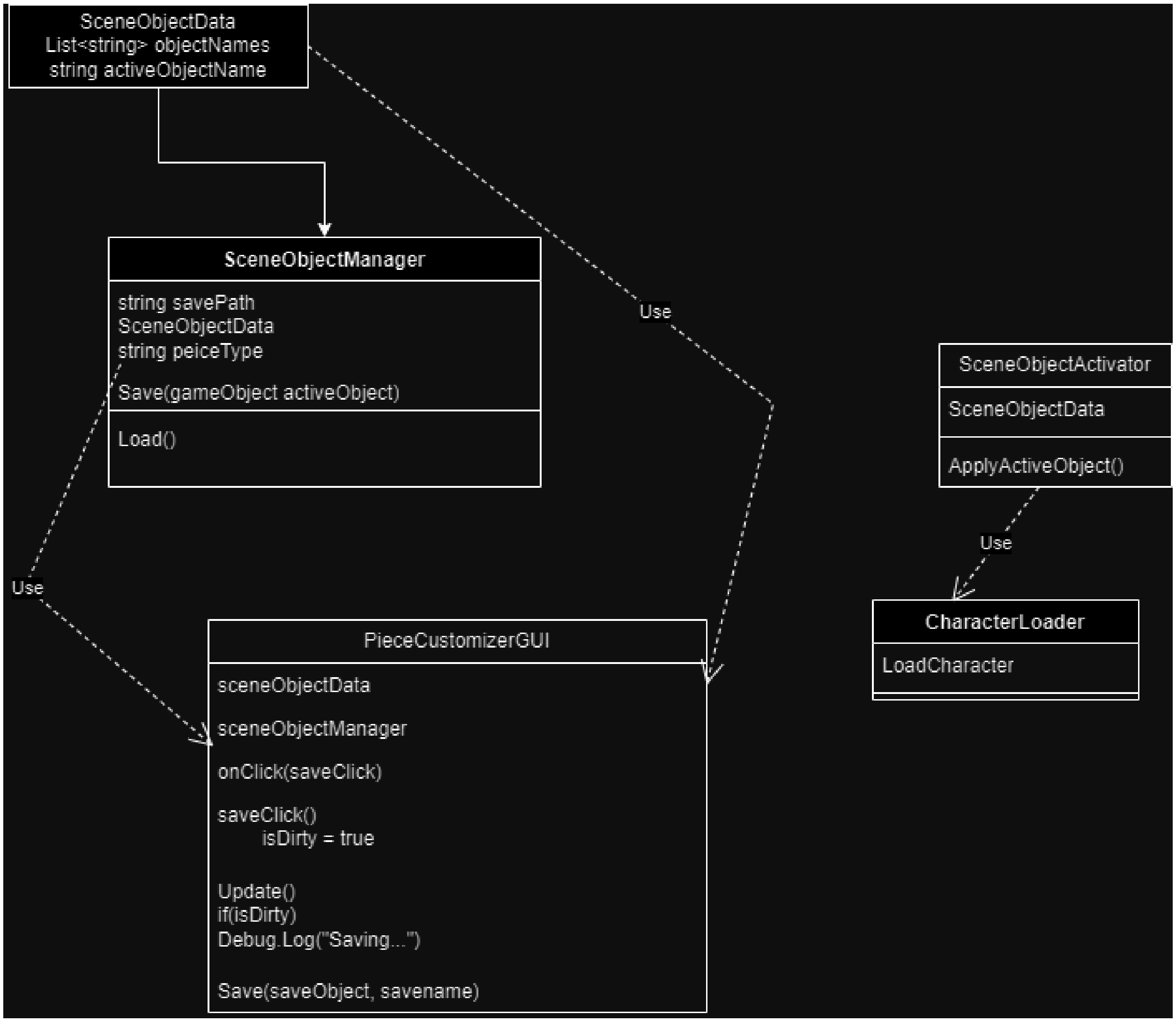


Explanation

Here is the object pool design pattern I used for our game. By the diagram it is simple, grab an object from the pool, use it, then return it to the pool and that is exactly how the object pool works. For our game the way I made it better using this design pattern was to add it to the players primary weapon since they will be using it the most. The first thing we do is edit our bullet class and give it the desired prefab. Next we feed that prefab into out projectile spawner, next the object pool creates a pool of these bullets that can be pulled from. We add an enqueue and dequeue function to change the next spawned bullet each time. Then our objectpool feeds into the gun script telling it how many bullets it has and how many are left. Without our object pool our gun would have to create a new bullet prefab every time it is fired, with our objectpool it really helps us optimize because we no longer need to create a destroy gameobjects every time the player shoots but instead we just get an object from the pool, this works especially well because there will be 3 players using this gun and that means the object pool is 3x effective. Without the object pool our networking would have to let every player know when a bullet is fired, where it goes and when it is destroyed. Now our server just has to relay the information of where the bullet goes, this is because the object pool only creates the bullet pool once, unlike creating it every time a bullet is fired.

Dirty Flag **[Michael]**

Diagram



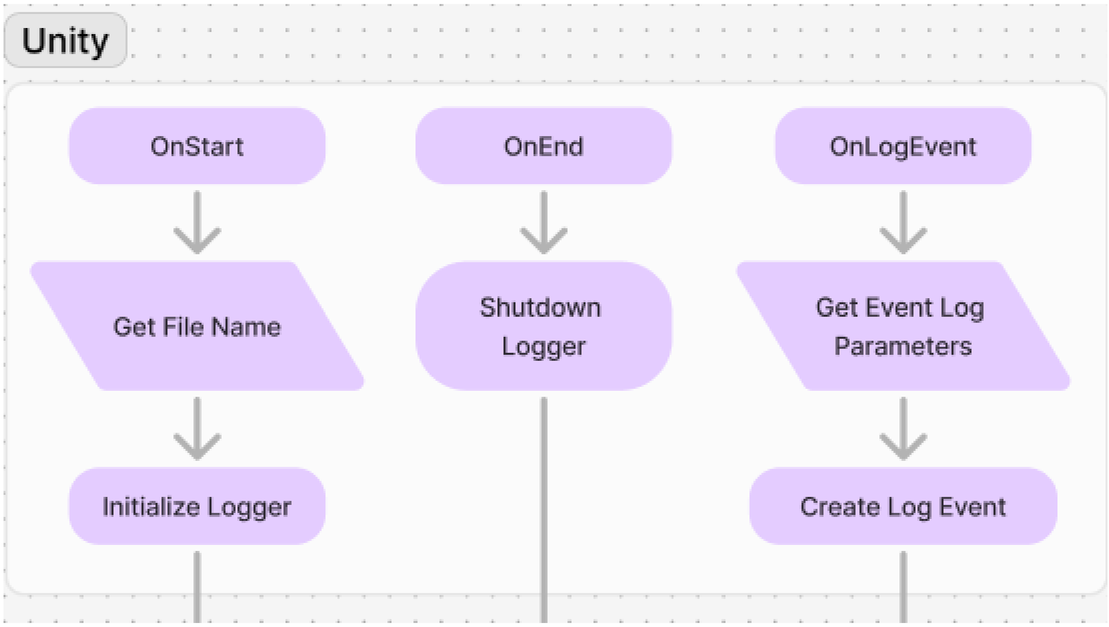
Explanation

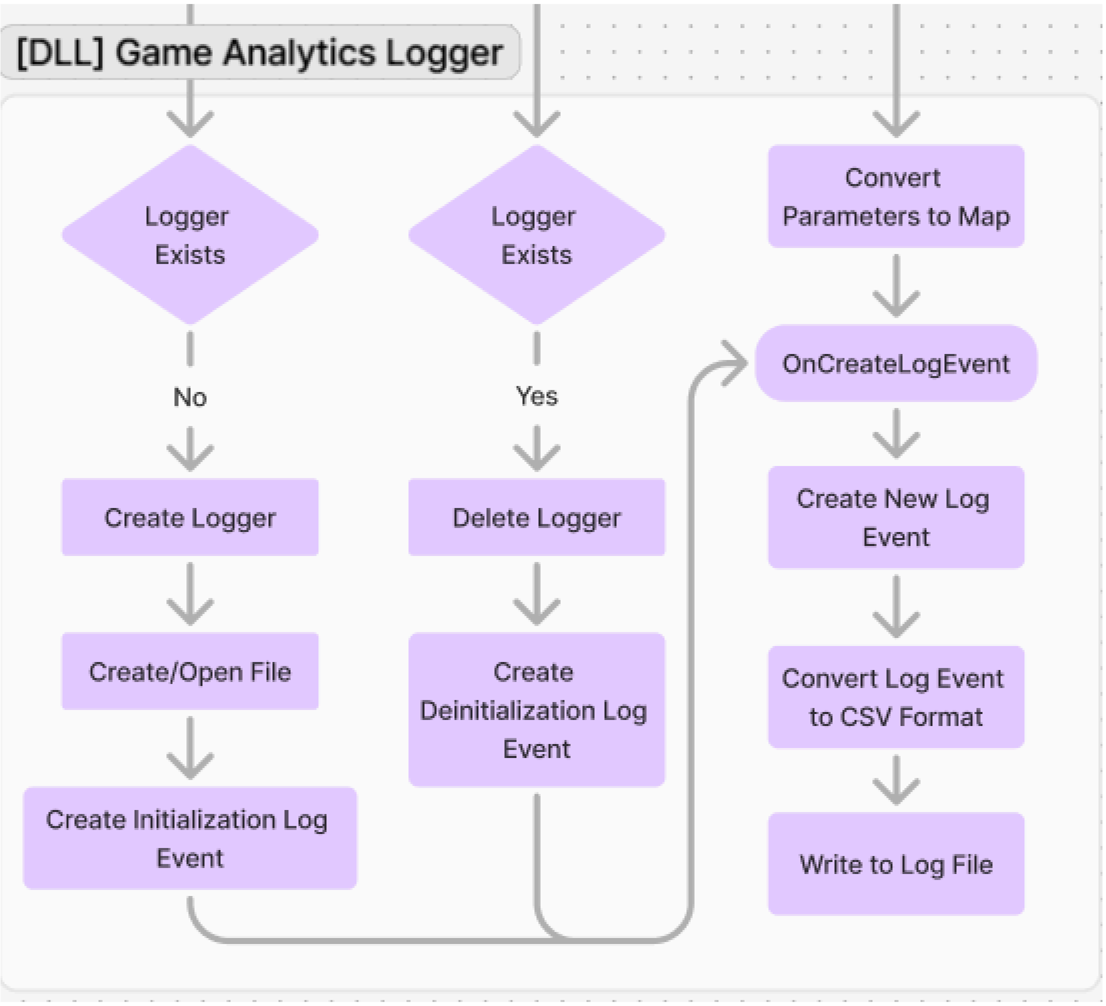
To figure out how I would Implement dirty flag, I first created the saving system for my game, and for the sake of the system to save consistently and keep the relevant information, but this posed a problem as these operations where being run too often so to optimize this, I added a dirty flag where the save function is called, to save the information only when the save button is pressed. The save function is implemented using a sceneObjectManager, in charge of saving the file to a json and loading from the json to a scriptable object. This allows the data to stay saved even if unity closes or fails, and allows me to load the data using the characterLoader script which references the activator and the manager to load the json into the scriptable object then loads the scriptable object data onto the player to select the active clothing.

Scenes required to play: customizer1, Loadchar

Plugin/DLL **(20/100)**

Diagram




[DLL] Game Analytics Logger
Logger Exists
No
Create Logger
Create/Open File
Create Initialization Log Event
Logger Exists
Yes
Delete Logger
Create Deinitialization Log Event
Convert Parameters to Map
OnCreateLogEvent
Create New Log Event
Convert Log Event to CSV Format
Write to Log File

Explanation

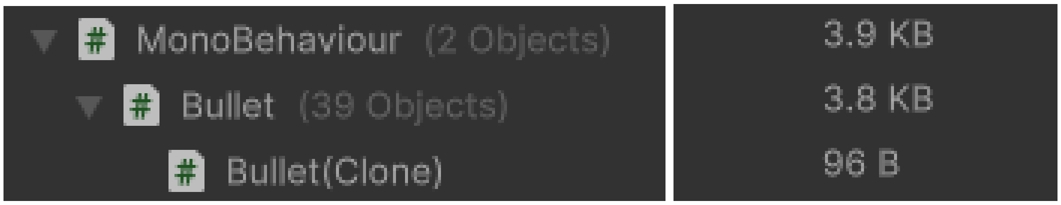
To create this dynamic linked library, we first had to brainstorm ideas as to where we could implement one. Initially, we were planning to implement the DLL with our game's procedural generation systems, however, the progress we already had working on the procedural generation, and the structure of the procedural generation classes made integrating the two undesirable. With more time, we could design a more open ended procedural generation system that would allow us to move it into a DLL. For now though, one of the next best ideas we had was for playtesting. Since our group's game has a lot of variables, namely, the procedurally generated map, the random monster mutation choices, and the items the players chose to use, it will be very hard to balance and make our game feel right. To that end, we need tools to help with making that process as achievable as possible. So, we decided to create a game analytics logger DLL. The goal of this DLL is to be generic enough to be used in other projects, but most importantly, allow us to log different events that happen in our game. This DLL allows us to specify an event name and unlimited event parameters and store them in a CSV file. The DLL is responsible for creating the log file and making edits to it. Unity calls the initialization and deinitialization, and provides the file location and events we want to log. By writing the logs into a file, we can have users that play our game from any location send us their log file, and get a ton of useful information. We can keep track of which mutations were picked, the game seed, when players died and how, or even which items players decided to craft. With this setup, we have defined a meaningful DLL that expands the playtesting/logging functionality of our game.

Performance Profiling **(20/100)**

Flyweight **[Denzil]**

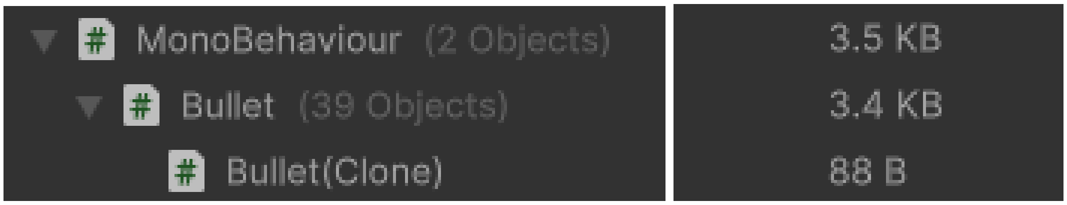
Measurements

*[**Without** the Flyweight Optimization Design Pattern]*



MonoBehaviour (2 Objects)	3.9 KB
Bullet (39 Objects)	3.8 KB
Bullet(Clone)	96 B

*[**With** the Flyweight Optimization Design Pattern]*



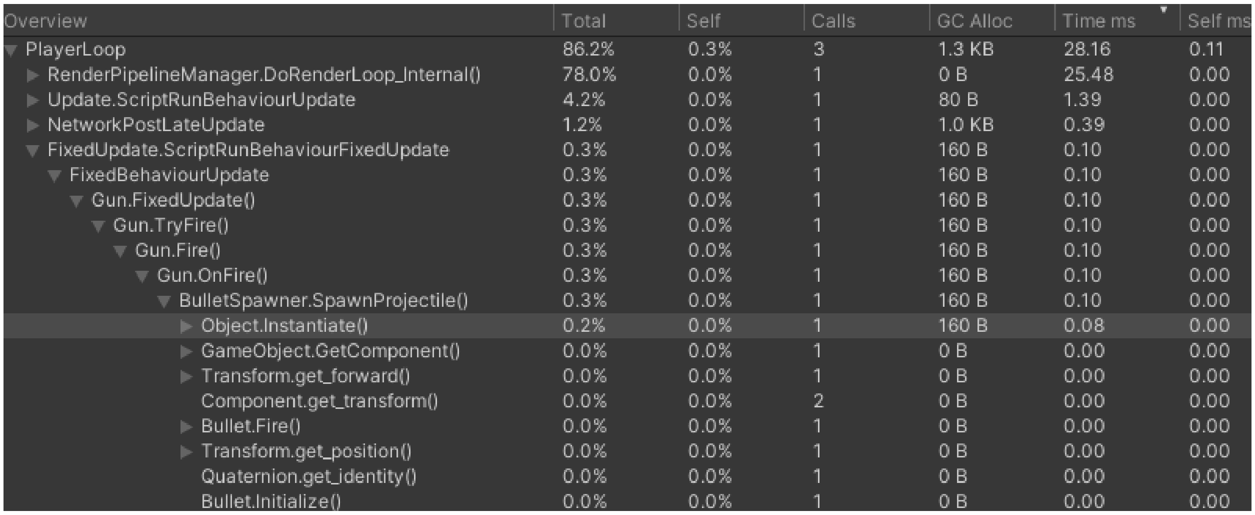
MonoBehaviour (2 Objects)	3.5 KB
Bullet (39 Objects)	3.4 KB
Bullet(Clone)	88 B

Explanation

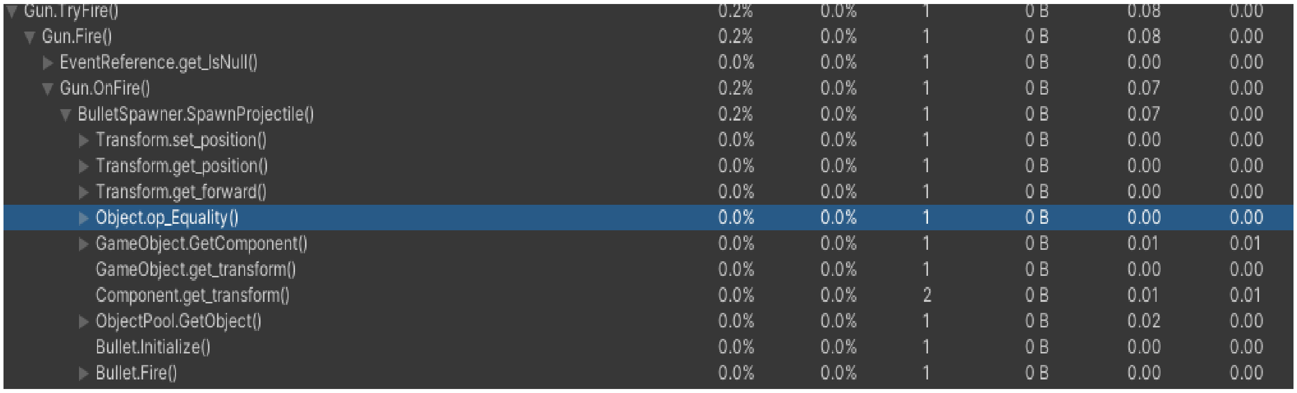
The two images above represent the measurements taken from the Memory Profiler in unity. The image on the top is the memory cost of one instance of the bullet class without the flyweight optimization design pattern implementation. The image on the bottom is that same cost, however with the flyweight optimization design pattern implemented. The difference of 8 bytes of information shows the current effect of the optimization. At this point, I would calculate the difference in memory using the size of the variables as reference, however, doing that would result in a difference of 12 bytes rather than 8. I'm not exactly sure why this is the case or what's going on in the background to cause this. Regardless, the calculation to get the difference of 12 bytes involves subtracting the cost of a pointer (8 bytes) from 4 floats and 1 integer (16 + 4 bytes).

Object Pool **[Aatif]**

Measurements



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	86.2%	0.3%	3	1.3 KB	28.16	0.11
RenderPipelineManager.DoRenderLoop_Internal()	78.0%	0.0%	1	0 B	25.48	0.00
Update.ScriptRunBehaviourUpdate	4.2%	0.0%	1	80 B	1.39	0.00
NetworkPostLateUpdate	1.2%	0.0%	1	1.0 KB	0.39	0.00
FixedUpdate.ScriptRunBehaviourFixedUpdate	0.3%	0.0%	1	160 B	0.10	0.00
FixedBehaviourUpdate	0.3%	0.0%	1	160 B	0.10	0.00
Gun.FixedUpdate()	0.3%	0.0%	1	160 B	0.10	0.00
Gun.TryFire()	0.3%	0.0%	1	160 B	0.10	0.00
Gun.Fire()	0.3%	0.0%	1	160 B	0.10	0.00
Gun.OnFire()	0.3%	0.0%	1	160 B	0.10	0.00
BulletSpawner.SpawnProjectile()	0.3%	0.0%	1	160 B	0.10	0.00
Object.Instantiate()	0.2%	0.0%	1	160 B	0.08	0.00
GameObject.GetComponent()	0.0%	0.0%	1	0 B	0.00	0.00
Transform.get_forward()	0.0%	0.0%	1	0 B	0.00	0.00
Component.get_transform()	0.0%	0.0%	2	0 B	0.00	0.00
Bullet.Fire()	0.0%	0.0%	1	0 B	0.00	0.00
Transform.get_position()	0.0%	0.0%	1	0 B	0.00	0.00
Quaternion.get_identity()	0.0%	0.0%	1	0 B	0.00	0.00
Bullet.Initialize()	0.0%	0.0%	1	0 B	0.00	0.00



Gun.TryFire()	0.2%	0.0%	1	0 B	0.08	0.00
Gun.Fire()	0.2%	0.0%	1	0 B	0.08	0.00
EventReference.get_IsNull()	0.0%	0.0%	1	0 B	0.00	0.00
Gun.OnFire()	0.2%	0.0%	1	0 B	0.07	0.00
BulletSpawner.SpawnProjectile()	0.2%	0.0%	1	0 B	0.07	0.00
Transform.set_position()	0.0%	0.0%	1	0 B	0.00	0.00
Transform.get_position()	0.0%	0.0%	1	0 B	0.00	0.00
Transform.get_forward()	0.0%	0.0%	1	0 B	0.00	0.00
Object.op_Equality()	0.0%	0.0%	1	0 B	0.00	0.00
GameObject.GetComponent()	0.0%	0.0%	1	0 B	0.01	0.01
GameObject.get_transform()	0.0%	0.0%	1	0 B	0.00	0.00
Component.get_transform()	0.0%	0.0%	2	0 B	0.01	0.01
ObjectPool.GetObject()	0.0%	0.0%	1	0 B	0.02	0.00
Bullet.Initialize()	0.0%	0.0%	1	0 B	0.00	0.00
Bullet.Fire()	0.0%	0.0%	1	0 B	0.00	0.00

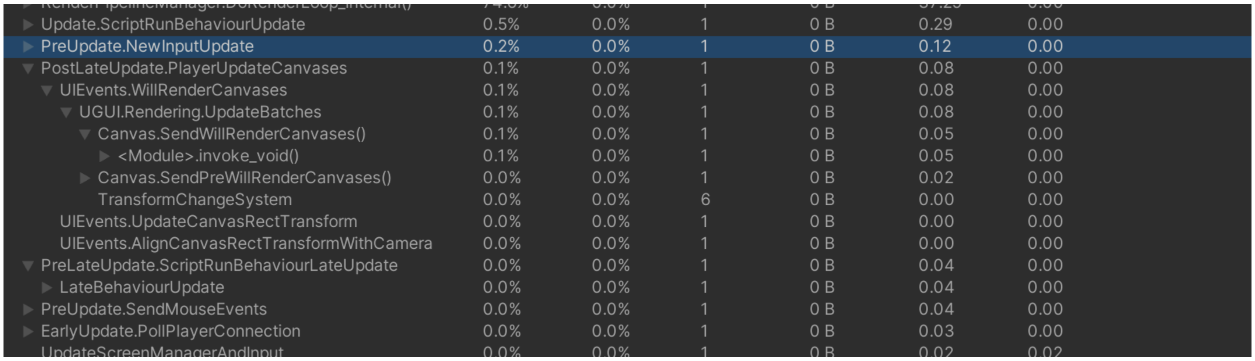
Explanation

The first screenshot is without object pooling, the second screenshot is with object pooling. The Gun.OnFire() method without pooling takes 0.10ms to work, with object pooling it takes 0.07ms. That is a 0.03ms difference every time the method is called which is often since it is a gun in a shooter game. Object pooling helped optimize here by reducing the amount of calls to the CPU. This is really effective for our game when it comes to optimization because it is a multiplayer game, by having out guns all now take 0.03ms less to compute it can travel the server faster and be replicated to other players sooner, without object pooling each players gun every single time is has been fired would take 0.10ms to work and then that information would need to be sent across. That means object pooling has worked to optimize our game by reducing the time it takes for the CPU to compute the required functions.

Dirty Flag **[Michael]**

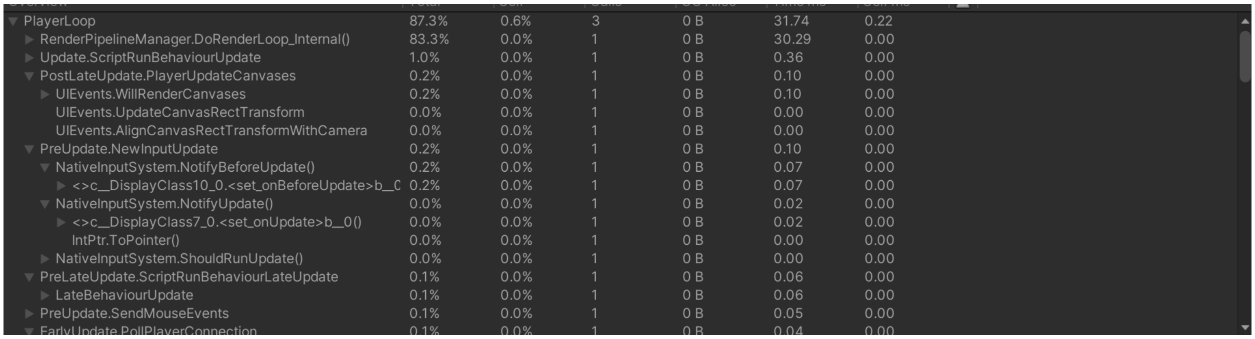
Measurements

With optimization



RenderPipelineManager.DoRenderLoop_Internal()	[illegible]	0.0%	1	0 B	[illegible]	0.00
Update.ScriptRunBehaviourUpdate	0.5%	0.0%	1	0 B	0.29	0.00
PreUpdate.NewInputUpdate	0.2%	0.0%	1	0 B	0.12	0.00
PostLateUpdate.PlayerUpdateCanvases	0.1%	0.0%	1	0 B	0.08	0.00
UIEvents.WillRenderCanvases	0.1%	0.0%	1	0 B	0.08	0.00
UGUI.Rendering.UpdateBatches	0.1%	0.0%	1	0 B	0.08	0.00
Canvas.SendWillRenderCanvases()	0.1%	0.0%	1	0 B	0.05	0.00
<Module>.invoke_void()	0.1%	0.0%	1	0 B	0.05	0.00
Canvas.SendPreWillRenderCanvases()	0.0%	0.0%	1	0 B	0.02	0.00
TransformChangeSystem	0.0%	0.0%	6	0 B	0.00	0.00
UIEvents.UpdateCanvasRectTransform	0.0%	0.0%	1	0 B	0.00	0.00
UIEvents.AlignCanvasRectTransformWithCamera	0.0%	0.0%	1	0 B	0.00	0.00
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.0%	0.0%	1	0 B	0.04	0.00
LateBehaviourUpdate	0.0%	0.0%	1	0 B	0.04	0.00
PreUpdate.SendMouseEvents	0.0%	0.0%	1	0 B	0.04	0.00
EarlyUpdate.PollPlayerConnection	0.0%	0.0%	1	0 B	0.03	0.00
UpdateScreenManagerAndInput	0.0%	0.0%	1	0 B	0.02	0.02

Without optimization



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	87.3%	0.6%	3	0 B	31.74	0.22
RenderPipelineManager.DoRenderLoop_Internal()	83.3%	0.0%	1	0 B	30.29	0.00
Update.ScriptRunBehaviourUpdate	1.0%	0.0%	1	0 B	0.36	0.00
PostLateUpdate.PlayerUpdateCanvases	0.2%	0.0%	1	0 B	0.10	0.00
UIEvents.WillRenderCanvases	0.2%	0.0%	1	0 B	0.10	0.00
UIEvents.UpdateCanvasRectTransform	0.0%	0.0%	1	0 B	0.00	0.00
UIEvents.AlignCanvasRectTransformWithCamera	0.0%	0.0%	1	0 B	0.00	0.00
PreUpdate.NewInputUpdate	0.2%	0.0%	1	0 B	0.10	0.00
NativeInputSystem.NotifyBeforeUpdate()	0.2%	0.0%	1	0 B	0.07	0.00
<>c__DisplayClass10_0.<set_onBeforeUpdate>b__C	0.2%	0.0%	1	0 B	0.07	0.00
NativeInputSystem.NotifyUpdate()	0.0%	0.0%	1	0 B	0.02	0.00
<>c__DisplayClass7_0.<set_onUpdate>b__0()	0.0%	0.0%	1	0 B	0.02	0.00
IntPtr.ToPointer()	0.0%	0.0%	1	0 B	0.00	0.00
NativeInputSystem.ShouldRunUpdate()	0.0%	0.0%	1	0 B	0.00	0.00
PreLateUpdate.ScriptRunBehaviourLateUpdate	0.1%	0.0%	1	0 B	0.06	0.00
LateBehaviourUpdate	0.1%	0.0%	1	0 B	0.06	0.00
PreUpdate.SendMouseEvents	0.1%	0.0%	1	0 B	0.05	0.00
EarlyUpdate.PollPlayerConnection	0.1%	0.0%	1	0 B	0.04	0.00

Explanation

As you can see, this is a frame after the button clicks on one with the optimization and without the optimization. The optimization reduced the amount of cpu being used for the script calls drastically, in Update.ScriptRunBehaviourUpdate, the number goes from 0.1% to 0.5%, this is a drastic improvement, as even the time taken to run has reduced from 0.36 to 0.12, this is a very slight but important improvement for the sake of optimization, where every millisecond can be used in other or more important operations.

Live Presentation **(10/100)**

Done in class…