MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

SUMMER PRACTICE REPORT

CENG300

STUDENT NAME: Batuhan Akçan

ORGANIZATION NAME: Beren Studio

ADDRESS: ETKİM Binası, Üniversiteler Mah. İhsan Doğramacı Blv, ODTÜ Teknokent D:31/21, 06800 Çankaya/Ankara

START DATE: 10/7/2023

END DATE: 18/8/2023

TOTAL WORKING DAYS: 30

STUDENT'S SIGNATURE **ORGANIZATION APPROVAL**

TABLE OF CONTENTS

1. INTRODUCTION

Beren Studio is a company that develops AR/VR applications and mobile games. The AR/VR applications that they develop consist of occupational health and safety applications, architectural applications, and games.

I have completed the first 5 weeks of the internship remotely, the last week was in-office. In the first 2 weeks of the internship, we were taught the Unity game engine (figure 1) and programming in C# language (figure 2). After that, we were divided into groups and each group were given a 3D game project. We as a group, developed our project in our personal computer throughout 3 weeks. In the last week, we uploaded our project into the company's computer and converted it into a VR project. Then, we tested the project in the company's VR headset (figure 3) and we have seen that it worked.

In this report, the project that I and my teammate Alpay Avşar developed is going to be explained in detail, then some information about the organization will be given. At the end of this report, a conclusion of my internship experience is going to be shared.

2. INFORMATION ABOUT PROJECT

2.1. ANALYSIS PHASE

The type of our game was first person shooter. We were given a game idea in which the player always runs forward and is able to move left and right, and shoots the enemies which attack him periodically. Also, the player needs to jump when he encounters an obstacle. There are loots on the road that give bonuses when they are collected by the player. If the player can successfully reach the end, he wins. If he dies, he loses.

2.2. DESIGN PHASE

In this phase, we have broadly designed the world. Initially, we placed the player into the world. We have made the camera child of the player. Then, we have made the gun child of the camera. Since the player is always going to go forward, we put a road in the space. We put to the sides of the road an empty terrain, then we placed several buildings on the terrain. After that, we placed obstacles on the road. Then, we put textures to the road and to the terrain. After that, we placed several enemies on the sides of the road, and on the buildings. We have placed lootbags randomly on the road. Lastly, we have put a finish line at the end of the road. After designing the world and developing all the gameplay, we designed the UI. We placed the health bar on the top-left of the screen and the score text just under it. Then we placed the *Game Over*! text and the *Play Again* button in the center of the screen as deactivated. We obtained player prefab, gun prefab, building prefabs, enemy prefab, obstacle prefab, and textures from Unity Asset Store (figure 4).

2.3. IMPLEMENTATION PHASE

Throughout the implementation phase, we have written several script files: *BloodSpray.cs*, *BloodSprayPool.cs*, *Bullet.cs*, *BulletPool.cs*, *BulletPoolEnemy.cs*, *Enemy.cs*, *Environment.cs*, *GameParams.cs*, *Gun.cs*, *LootBag.cs*, *PlayerController.cs*, *PlayerStats.cs*, *TagHolder.cs*, *UIController.cs*. I am going to explain what we have implemented in these script files in the following paragraphs.
Firstly, we knew that we need to be able to control the character. So we wrote the *PlayerController* script, in which we implemented *CursorLockUnlock* function that locks the cursor when the user hits *Escape*, and unlocks it when he/she hits it again. After that, we implemented *CameraControl* function which takes Mouse X and Mouse Y as input, multiplies them by *MouseSensitivity*, and assigns it to the local rotation of the camera. Then, we implemented *Move* function which takes *A* and *D* keys as input and moves the player to the right and to the left. Also, takes *Space* key as input and makes the player jump, with a constant jump speed. We added a constant velocity pointing forward, to move the player forward without pressing any key. Lastly, we implemented the *Gravity* function, that checks if the player is grounded, if he is not, then applies gravity. Gravity is obtained by multiplying the mass of the player by gravitational acceleration. Then it is added as a force to the player gameobject.
Secondly, we needed to implement shooting mechanism. For this problem, there are 2 solutions: Raycasting and pooling system. Raycasting is easier to implement but costlier, pooling system is more complicated but more efficient. Also, in raycasting, there is actually no bullets, so you can not see the bullets while you are firing. Since this game will eventually become a VR game, our project manager wanted us to make the bullets visible. So we have chosen pooling system. In pooling system, you create a pool of objects (bullets in this case) and put them in a queue as deactivated. When the player fires, the next bullet is dequeued, activated and fetched by the *Gun* script. After the bullet collides to anywhere (walls, enemies) or its lifetime ends (controlled by coroutines) it is deactivated and enqueued again to the queue. We have implemented these in *BulletPool* script. Then, in *Gun* script, when the user clicks *LMB*, the next bullet is fetched by *GetBullet* function of *BulletPool* script.
After that, we wrote the *Bullet* script, in which there were *PlayerShoot* and *EnemyShoot* functions. *PlayerShoot* is called by the *Gun* script, if the fetched bullet is not null. This function is responsible for assigning the bullet's position, rotation, velocity, etc. according to the gun's current position and rotation. Also, the coroutine that inactivates the bullet after its lifetime ends is called in this function. *EnemyShoot* function is similar, but it has different parameters and is called by *Enemy* script.

In the *Enemy* script, we firstly implemented the *Shoot* function. In that function, we first calculated the distance between the enemy and the player. If distance is less than or equal to the enemy attack range, the enemy's rotation is arranged so that it looks towards the player. Then we made the enemy shoot periodically as follows: If *attackTimer* $<= 0$, call the *Shoot* function, and assign *attackDelay* to *attackTimer*. Else, subtract the time passed since the beginning of the game from *attackTimer*. In the *Shoot* function, the shooting direction is calculated and passed to the *EnemyShoot* function in the *Bullet* script. Also, there was *OnCollisionEnter* function that checks if the enemy is collided by a bullet, if it is, firstly it stores the collision point in a variable which will be used by the *BloodSprayPool* script. Then, it calls $TakeDamage$ function, in which enemy's $currentHealth$ is decremented by an amount of *playerDamage*, if the health goes below zero, then the enemy gameobject is destroyed and player's score is incremented by 1 via the getter and the setter of the score in the *PlayerStats* script. In the *OnCollisionEnter* function, if a bullet hits an enemy, bullet is immediately destroyed without waiting the end of its lifetime, by calling the *Deactivate* function in the *BulletPool* script. Also, in the *Environment* script, *Deactivate* function is called when a bullet hits the ground or buildings. Lastly, there was the *Lerp* function. There were certain enemies chosen by us that are going to go between two points (point A and point B for instance). We implemented it with *Lerp* function. There was a field *timeElapsed* which was equal to zero initially. If $timeElapsed < lerpingDuration$, enemy goes from point A to point B in $\frac{timeElapsed}{lerpingDuration}$ seconds and the passed time is added to *timeElapsed*. Else if $timeElapsed < lerpingDuration * 2$, enemy goes from point B to point A in $\frac{timeElapsed - lerpingDuration}{lerpingDuration}$ seconds and the passed time is added to *timeElapsed*. Else, we restarted the lerping, that is, zero is assigned to *timeElapsed*.
PlayerStats is the script in which *health*, *score*, *isDead* fields of the player are held. The *score* field has a getter and a setter, which are called by the *Enemy* script. The *isDead* field has only getter but not a setter, because the player should only die within *PlayerStats* script. The getter of *isDead* is used by several scripts to make the player move, control the camera, make the enemies be able to shoot, etc. if and only if the player is not dead. There was *OnCollisionEnter* function which checks whether the player collides with a wall or a bullet, and decrements *health* by *wallDamage* or *bulletDamage*. If it is a bullet, it is immediately deactivated and enqueued to the bullet queue by the *Deactivate* function of *BulletPool* script. In this function, if $health <= 0$, then *isDead* becomes *true*. There was also another function, $OnTriggerEnter$. In it, whether the player triggers a health loot or a damage loot is checked. If a health loot is triggered, then *health* is incremented by *healAmount* and the health loot gameobject is destroyed. If a damage loot is triggered, then for all enemies, $damageTaken$ is incremented by $damageBuff$ using the getter and the setter of $damageTaken$. Then the damage loot is destroyed.

When an enemy is hit, there needed to be a blood spilled from the enemy. Since the player can hit multiple enemies in a short range of time, blood should be spilled from all of them. Thus we needed another pooling system in which blood spray effects are held in a queue. So we have implemented *BloodSprayPool* script. We have taken the blood spray effect prefab from Unity Asset Store. Like the *BulletPool* script, a certain amount of blood spray effects are enqueued as inactive when the game starts. When the player hits an enemy, in the *OnCollisionEnter* function of that enemy's *Enemy* script, the contact point of the bullet is stored and then *GetBloodEffect* function of *BloodSprayPool* is called, which dequeues the first blood spray effect. Then, if the effect is not null, *InstantiateEffect* function of *BloodSpray* script is called, which takes the contact point as argument and assigns it to the effect's position, activates the effect, and then calls the coroutine *DeactivateEffect* which deactivates the effect after its lifetime ends.
We encountered a problem in which when the player and enemies try to get a bullet from the pool simultaneously, neither of them could get the bullet. In order to solve this problem, we implemented seperate pools for each of them. *BulletPoolEnemy* is completely the same as *BulletPool*, but it keeps track of the bullets fired by enemies, whereas *BulletPool* keeps track of the bullets fired by the player.
The pooling amounts of all pools were assigned as minimum needed amount that we have calculated beforehand by taking into account the fire rate of the player and enemies, the lifetime of a bullet, etc.
The coroutines used in pools (e.g; *DeactivateBullet*, *DeactivateEffect*) have a similar syntax: First, they *yield return* the delay (which can be the lifetime of a bullet, or lifetime of an effect), then, the effect or the bullet is deactivated and enqueued back to its queue.
GameParams was the script in which we stored all constant parameters used by all scripts. They were stored in a *static class*, with *public const* keywords.
In *TagHolder* script, we stored all tags by using the keywords *public const string*. Tagging is a feature developed by Unity. By means of that feature, one can tag the objects that are going to behave similarly and can reach all the objects that have the same tag.
GameParams and *TagHolder* scripts have improved the efficiency a little more because we assigned constant values once, and then used them anywhere.

Then we have implemented animations. An animation is a move that an object performs which starts if a certain condition is fulfilled, and ends after some time (usually a few seconds). Animations are implemented using finite state machines, which is a feature of Unity game engine. In that finite state machine, there is an entry state and an exit state. The states, which are the animations, and the transitions between them, which are the conditions, are created by the developer. The state machine is called an animator. In our project, there were 2 animators: One for the player, and the other for enemies. In the enemies' animator (figure 5), when the game starts, the animator transitions to the *Idle* state. When an enemy attacks, the animator transitions to the *Attack* state. After its attack is completed, it transitions back to the *Idle* state. When an enemy is killed by the player, the animator transitions to the *Exit* state no matter what its current state is, which is the final state. In the player's animator (figure 6), when the game starts, the player starts to run and thus the running animation is played in a loop. Since we did not have the animation for simultaneously shooting and running, we did not implement a shooting animation. Nevertheless, there was the *Laying* animation which is activated when the player dies. Also, the *Idle* animation was the animation that is played when the player reaches the finish line. When the duration of *Idle* or *Laying* animations end, the animator transitions to the *Exit* state. The animations that were used in the project were obtained from Unity Asset Store.
Finally, we implemented the looting system. There were lootbags that were positioned randomly in the game. The lootbags all had *currentEndurance*, which specifies the number of times that the player needs to hit the lootbag to collect the loot in it. In the *LootBag* script, there was the function *OnCollisionEnter*, which checks if the lootbag is hit by a bullet, if it is, then decrements *currentEndurance* by 1. If *currentEndurance* becomes zero, the lootbag is destroyed and *InstantiateLoot* function is called with the parameter *spawnPosition*, which specifies where the loot is going to spawn and is equal to the position of the lootbag. The *InstantiateLoot* function firstly calls *GetDroppedItem*. *GetDroppedItem* randomly selects a number between 0 and 100, if it is between 0 and 20, the function returns the damage loot prefab, else it returns the health loot prefab. So the dropping chance of the damage loot was %20, the health loot was %80. After that, *InstantiateLoot* function checks whether the returned value is null, if it is not, it calls the built-in function *Instantiate* which spawns the selected prefab at *spawnPosition*. When the player walks into the spawned loot, the *OnTriggerEnter* function of *PlayerStats* is called automatically.

After the gameplay is finished, we implemented the UI. UI stands for User Interface, which can show the health of the player, the stamina of the player, the score of the player, currently selected gun, the ammo amount in the magazine, etc. on the screen during the gameplay. In our game, there were just the health and the score. We designed the UI which was explained in the design phase of the report in detail. In the *PlayerStats* script, when *health* is changed anywhere, the *SetHealth* function of *UIController* script is called with the parameter *amount*, which is the *health* after the change. This function fills the health bar between 0 and 1. Since *health* is between 0 and 100, it divides *amount* by 100 and then assigns it to the *fillAmount* field of *healthBar* gameobject. Similarly, when *score* is changed anywhere, the *SetScore* function of *UIController* is called with the parameter *amount*, in the function *amount* is assigned to *scoreText*, which shows the score in the screen.
There was a finish line at the end of the road. When the player walks on it, in the *OnTriggerEnter* function of *PlayerController*, whether the triggered object was the finish line is checked, if it is, then *isFinished* flag becomes *true* and it stops the game (stops the player, inhibits the camera control, prevents enemies to attack, etc.). Then, *ActivateEndGameUI* function of *UIController* script is called in which *Game Over*! text and *Play Again* button are activated. If the player dies before reaching the finish line, the function is also called. When the user clicks the *Play Again* button, *PlayAgain* function of *UIController* is called which reloads the game scene.

2.4. TESTING PHASE

After the project was completed remotely, we went to the office of the company and uploaded the game into one of their computers. We converted the desktop game into a VR game and made some small changes in the game such as: We arranged the position of the camera, arranged the position of the gun such that it fits into the user's hands, changed the inputs from keyboard and mouse to VR controller, changed the position of UI from top-left to the arm of the user so that he/she sees his/her health and score by rotating his/her arm, etc. After that, we built the game for Android (figure 7), and tested with a VR headset. We saw that values of some game parameters (e.g; *speed*, *playerBulletSpeed*, *fireRate*) need to be changed so that the game becomes more enjoyable for VR players. We changed them and re-built the game. We tried the game and saw that the game was successfully completed and was ready to play.

3. ORGANIZATION

Beren Studio (figure 8), which is established in 2020, is a company that develops AR/VR applications and mobile games. The shareholders of the company are: Ali Rauf Kolbakır (CEO), Dr. Ömer Ekmekçi (CTO), and Dr. Özgür Kaya (Consultant). Dr. Ömer Ekmekçi completed his PhD degree at METU Computer Engineering department with a specialization in artificial intelligence. Dr. Özgür Kaya also completed his PhD degree at METU Computer Engineering. Ali Rauf Kolbakır has a BSc degree at OMU Civil Engineering department. The company currently has 16 employees, who consist of Unity developers, 3D artists, and level designers.

According to their web page (beren.studio), they have developed 20 VR projects, 5 AR projects, 5 mobile games, 1 AI project, and 2 Metaverse projects. VR projects consist of Occupational Health and Safety (OHS) applications, architectural applications, and VR games. The most of the OHS applications were developed for Labor and Social Security Training and Research Center (ÇASGEM) and DAI Global, which is one of the world's leading development companies. Some of their OHS applications are: Coal Mining OHS VR Training Simulation, Construction OHS VR Training Simulation, Oil & Gas Facility OHS VR Training Simulation, Manufacturing OHS VR Training Simulation. They are also developing a deep learning R&D project with METU Computer Engineering department. The purpose of this object is to create three dimensional mesh objects from text with deep learning methods. They have recently released their first VR game Tales of Tsynara on Oculus App Lab and several VR games are currently being developed.

Since AR/VR technologies have become more and more popular in the last few years, it is foreseeable that the companies working on AR/VR are going to grow swiftly in the near future and Beren Studio seems to be one of them.

4. CONCLUSION

The internship was a great experience for me. I have learned game development with Unity game engine, the basics of C# programming language, and the specifics of VR development. Also, I have improved my knowledge about data structures and object-oriented programming paradigm. I have been to Beren Studio's office and seen how an office environment is. This was the first time in my life that I have been to an office. It was also the first time that I used a VR headset, which was an unprecedented experience.
Game development is an exciting occupation and this internship made me consider to continue my career in game sector.

5. APPENDICES

Figure 1: Unity is a cross-platform game engine that is used to create 2D & 3D games, as well as simulations

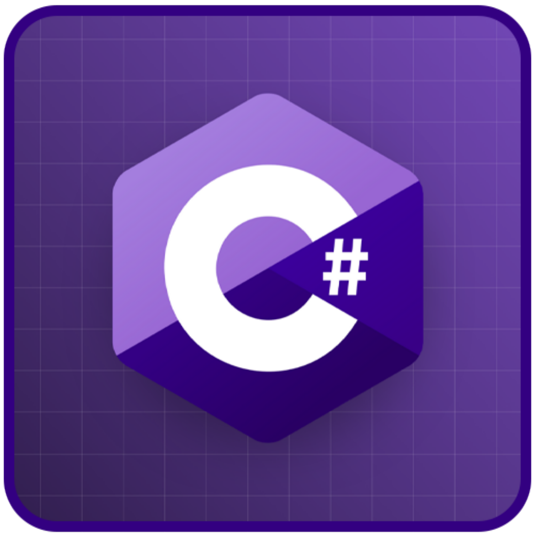
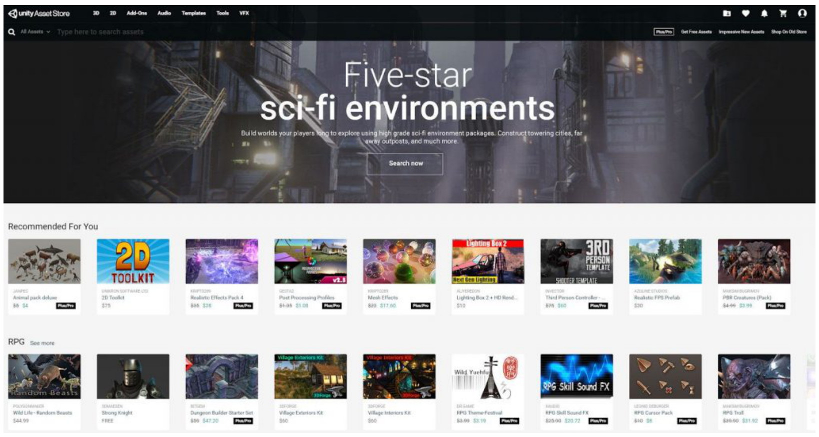


Figure 2: C# is a high-level programming language created by Microsoft that supports multiple paradigms

Figure 3: A VR headset with its controllers



Build worlds your players long to explore using high grade sci-fi environment packages. Construct towering cities, far away outposts, and much more.

Figure 4: Unity Asset Store is a store where prefabs, textures, effects, animations, etc. are sold

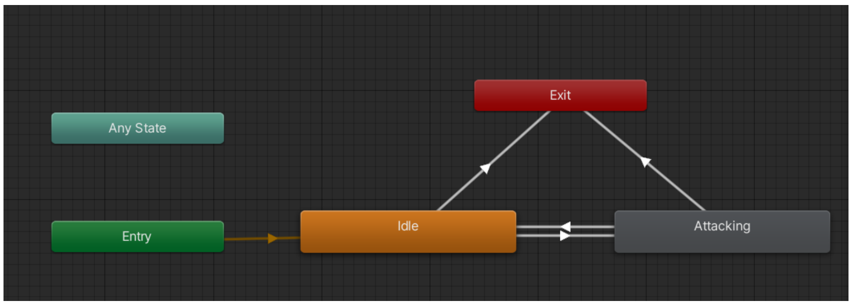


Figure 5: The animator that controls enemies' animations

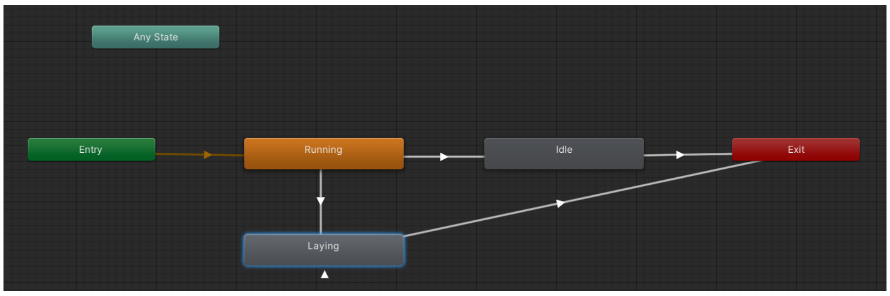


Figure 6: The animator that controls player's animations

android

Figure 7: Android is an operating system that runs on some VR devices, and also mobile phones and tablets

BEREN STUDIO

Figure 8: Beren Studio