

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1040

**SUSTAV ZA PRAĆENJE UTJECAJA
STRESNOG OKRUŽENJA RAČUNALNE IGRE
NA KOGNITIVNE SPOSOBNOSTI ČOVJEKA**

**SYSTEM FOR MONITORING THE IMPACT OF
STRESSFUL COMPUTER GAME
ENVIRONMENT ON HUMAN COGNITIVE
CAPABILITIES**

Ana Stepić

Zagreb, June 2013

Zagreb, 2. ožujka 2015.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1040

Pristupnik: **Ana Stepić (0036448340)**

Studij: Računarstvo

Profil: Računarska znanost

Zadatak: **Sustav za praćenje utjecaja stresnog okruženja računalne igre na kognitivne sposobnosti čovjeka**


Opis zadatka:

Proučiti razvojno okruženje za računalne igre Unity te povezivanje s uređajima Oculus Rift i Delta Six. Posebno razraditi povezivanje navedenih uređaja u jedinstveno okruženje. Sagraditi scenu koja će uključivati prikaz proizvoljnih slika i video-sekvenci te animiranih ljudi. Ostvariti mjerenja odaziva pri realizaciji pojedinih aktivnosti u definiranom okruženju i time realizirati svrhovitu računalnu igru (engl. serious game). Računalna igra treba omogućiti praćenje utjecaja stresnog okruženja same igre ili eventualnih vanjskih poremećaja na brzinu odziva čovjeka, razlikovanje poželjnih i nepoželjnih objekata i snalaženja u takvom okruženju. Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 13. ožujka 2015.

Rok za predaju rada: 30. lipnja 2015.

Mentor:


Prof. dr. sc. Željka Mihajlović

Djelovođa:


Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:

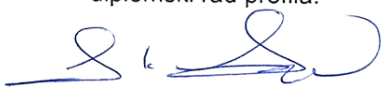

Prof. dr. sc. Siniša Srblić

Table of Contents

1. Introduction	5
2. Application Design	7
2.1. Tutorial.....	8
2.2. Main Level	9
3. Development.....	9
3.1. Unity	9
3.2. Main Gameplay Implementation	10
3.3. User Interface & Input Implementation	11
3.4. Audio Playback	13
3.4.1. Dialogue	14
3.5. Tutorial Scene.....	15
3.6. Main Gameplay Scene	20
3.6.1. Avoiding user learning by randomization.....	21
3.7. Integration with Oculus Rift	26
3.8. Integration with Delta Six	28
3.9. Increasing Graphical Fidelity.....	30
3.10. Performance Improvements	33
4. Stressors.....	37
4.1. Image Interruptions.....	38
4.2. Video Interruptions.....	38
5. User Performance Measurement.....	39
6. Future Works	40
7. Conclusion	43
8. Sažetak.....	44
9. Abstract	45
10. References.....	46

11. Abbreviations.....	49
12. Appendix	50
12.1. List of Used Third Party Assets.....	50
12.2. Software Usage Instructions	53

1. Introduction

Game development industry is a rapidly evolving and blooming industry with large impact on economy - in 2014 PC and console game revenue exceeded 46 billion U.S. dollars [1]. With inclusion of the growing mobile game industry the total revenue of the game industry in 2014 is 81 billion U.S. dollars [2] [Figure 1], which far exceeds movie entertainment global box office revenue in 2014 which was just 36 billion U.S. dollars [3].



Figure 1 - Global game revenues in 2014

However games are more than just a tool to provide entertainment, they also have a great potential to help us in numerous ways and are already being used in various different fields such as health care and education [Figure 2]. Whether it is training personnel, teaching specific knowledge and skills in an easy to remember and learn way or rehabilitation and helping impaired individuals – games have the potential to help.



Figure 2 - Usage of serious games in various fields and industries

Rapid evolution of the computer hardware industry had now made possible what game enthusiasts had been dreaming for years: appearance of consumer grade VR (virtual reality) equipment whose utility is not just limited by conventional games.

This thesis looks into development of a serious game the purpose of which is not to entertain but rather to help - a game that uses the potential of VR to help soldiers in various ways. It can provide a safe environment to train and educate them and offer valuable insight into improving their performance. It could even help them deal with stressful situations and stressors (stress causing events) and potentially help in curing and dealing with PTSD (Post-traumatic stress disorder) ideally even preventing PTSD from ever being developed.

The purpose of this thesis is in creating a basic FPS (first person shooter) game utilizing available VR technology: Oculus Rift HMD (head mounted display) and Delta Six gun controller to create a virtual battle experience that would allow measurement of performance and impact of simple stressors on cognitive performance.

2. Application Design

The application is supposed to be intuitive to use and have a realistic graphics and gameplay. Realistic graphics will help rise immersion and player's sense of presence inside the virtual world. It is supposed to utilize available VR devices to further increase the immersion and make the experience as close to reality as possible.

Target environment of the game is supposed to be based on Afghanistan desert [Figure 3]. Models and textures needed to make such an environment were not created as a part of the project, rather, a fitting environment was obtained through Unity Asset store that already contained required assets.



Figure 3 - Dashti Margo desert in Afghanistan

To ensure users do not have to read the manual to learn how to play, a special tutorial level needs to be implemented that will teach new players everything they need to know about the application. Tutorial level is also supposed to help players get used to the application so that this initial adaption period would not impact performance measurements.

For users that know how to play, the application is supposed to provide an environment where they are subject to conflict with enemies and emotional stressors. In such an environment soldiers are supposed to train to increase their resistance to emotional stressors while engaging in simulated combat situations.

To provide an adequate experience, the gameplay should contain basic elements:

- Movement in all directions
- Jumping over obstacles
- Shooting Delta Six rifle
- Using the scope on the rifle to more accurately shoot
- Being able to reload the rifle when it runs out of ammo
- Providing a way to navigate through the game without a need for extra controllers besides Delta Six.
- Being able to damage enemies and allies
- Taking damage if shot.

2.1. Tutorial

Tutorial level is supposed to be a short and simple level that teaches players everything they need to know and helps them get used to the game before tackling the main level. It should educate players in a straightforward easy way not requiring previous experience with games or reading the application manual. To achieve that, a small linear level was designed that doesn't allow players many options, but they are instead supposed to follow the path the game asks them to.

Best way to communicate information to the player is to ask them to do things while providing information through audio. That is why tutorial level was designed to have informational text and dialogue spoken to the player.

The tutorial should teach things in an appropriate order, from basic to advanced. Decided teaching order is the following:

- Looking around and rotating the player character
- Moving the player character
- Aiming and shooting
- Using the scope of the Delta Six rifle to make more accurate shots in game

- Reloading the riffle
- Jumping
- Recalibrating the gun controller
- Using learnt knowledge in a fight with enemy agents

2.2. Main Level

Main level is supposed to be designed in a way that allows players to take any route they wish without limiting them on just a few available paths. For this reason it was decided this would be an open area level, unlike the tutorial which was guided and intended to have just one path.

It is also supposed to be large enough to not be completed too quickly and small enough to not decrease performance of the application. It should contain a fair amount of enemies and allies with varying behaviors.

To minimize effects of user learning caused by repetitive gameplay, main level needs to include randomization mechanisms to minimize repetition.

Unlike the tutorial level, the main level needs to collect as much performance data as possible and write it out to disk for later analysis. It should also include emotional stressors that can be enabled or disabled.

3. Development

3.1. Unity

The goal of the project was to make a product with available tools, using existing packages and assets together into a whole complete product. If no tools were available a project of this scope could have never been achieved in amount of time it had to be done in.

Therefore, a number of existing applications and tools needed to be used to make this project a reality – the most important tool being a game engine.

Game engine is a software framework designed for the creating and development of video games. It handles essential tasks such as animation, rendering and physics.

The engine chosen for this project was Unity because its ease of use and availability of documentation, plugins and assets needed for this project.

3.2. Main Gameplay Implementation

To reduce the amount of work needed to be done to implement required functionality, UFPS package was used. UFPS is a Unity plugin that has base FPS game mechanisms already implemented. By reusing existing functionality, the development efforts were redirected elsewhere which ultimately resulted in a bigger and better application.

As a part of UFPS package, a first person character controller was included along with the models of rifle and body of the character. Required animations were also included, along with basic mechanisms such as health and ammo control, movement, shooting, jumping and reloading.

Although those mechanisms were already provided by the UFPS package, they had to be set up and modified through Unity Editor to achieve the right experience for this particular application.

There were also some features that were a part of the set up that were creating issues and had to be manually removed. Such a feature was shaking of camera with player movement. When people walk or run their head moves up and down slightly, which is what developers of UFPS included as a part of UFPS to enhance the realism of FPS games. However, if same thing happens in VR, the camera moves up and down but our real body in the real world does not. This creates conflict between virtual world and reality that breaks immersion and realism and ultimately causes motion sickness in players. It also makes it very difficult to aim with the gun controller because of the unpredictable shaking of camera.

Another problematic feature was low accuracy of the rifle. In the initial set up, a rifle had a certain radius of accuracy – when the player aimed at one point, chance of him hitting that exact spot were very slim. When the player fired the rifle, the system found a random point in the circle around the point where the gun is aiming. The lower the accuracy, bigger radius of the circle. That random point was then chosen as the place to fire the shot at.

This behavior was obviously something not realistic. Real life guns do not have a randomized hitting behavior. Accuracy of the shot depends on the skill of the soldier firing it. Therefore, this randomized shooting was removed.

There were also missing features of the base gameplay that had to be implemented, such as sniping mode and creation of hittable targets.

Inside UFPS, usage of the scope was implemented as an animation that changed the position of the rifle, however, the zooming feature of the scope was not implemented – it needed to be implemented manually.

A special sniping script was written and attached to the weapon which detects when sniping mode is activated and changes camera field of view to create a “zooming in” effect.

Implementation of hittable targets required implementing the method called Damage which UFPS calls when something gets hit. Each target can have its own specific logic to perform upon being shot.

3.3. User Interface & Input Implementation

Controlling of the application is done by using keyboard and mouse. Each action the player can do is mapped to a certain keyboard key using Unity Input. The exception are control of the camera and aiming – camera is directly controlled by HMD, while aiming is done with the mouse (or rather, the Delta Six rifle).

Through the application, special attention was made to minimize the need for a graphical user interface. Information is communicated to player primarily through audio. However, there are certain situations where graphical interface was necessary. For those scenarios, uGUI was used – Unity’s new gui system that was first introduced in Unity 4.6.

uGUI button components were used to create initial menu that allows the user of the application to choose whether to play tutorial or main level, and whether to play with or without stressor interruptions.

uGUI text and sprite background was used to create windows that show messages to the player. In tutorial scene, it was necessary to teach the player how to play. While most information is conveyed with audio dialogue, some of the most

important text is also written in little popup windows, in case the player fails to understand what had been said [Figure 4].



Figure 4 - pop up window teaching the player about game play

A window that appears when the player finishes the level or pauses the level was similarly done. Pause window shows information about remaining enemies and gives player choice to exit the game or switch scene [Figure 5]. The end level window provides similar choices.



Figure 5 - Pause Window

All GUI windows were animated so that windows pop up onto the screen. Animation of GUI components was done by LeanTween plugin. LeanTween is a

plugin that provides C# methods to animate properties such as position, scale and opacity over time.

3.4. Audio Playback

Audio is an integral part of the realistic experience and as such, special attention was put into it. Unity 5 audio mixer was used to adjust audio volumes of different audio groups used through the project. Used audio groups are the following:

- Voice – for dialogue
 - Important - for instructions on how to play
 - NonImportant
 - Player – player character commenting current state of the game e.g. mentioning he is out of ammo
 - Background – enemy shouts and conversations
- Sfx – sound effects such as gunshots, footsteps, etc.
- Ambient – for ambient effects and music such as sounds of the desert wind blowing.

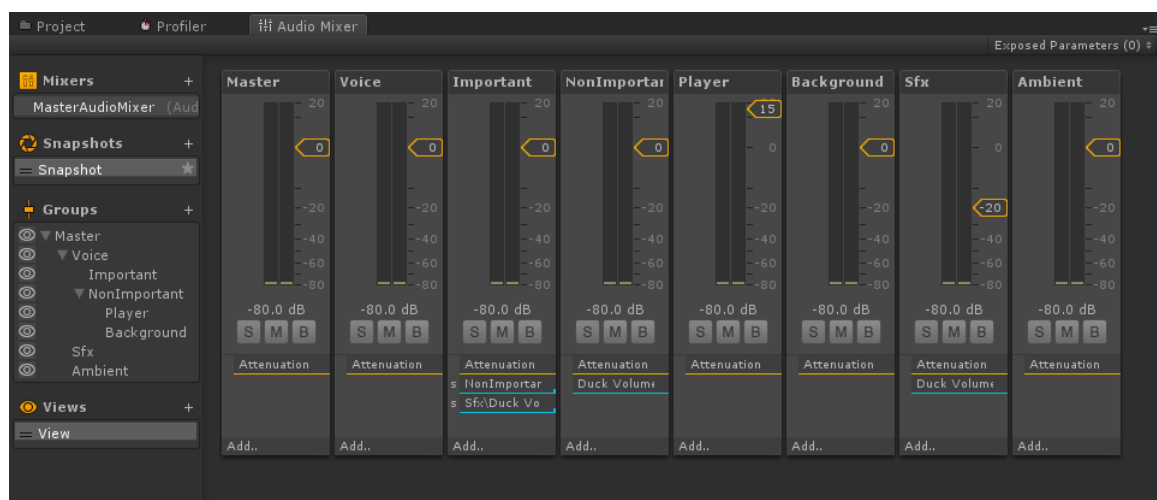


Figure 6 - Audio Mixer

Audio mixer special effects were used to achieve ducking effect – when important dialogue takes place, irrelevant noises such as ambient noise and special sound effects (e.g. gunshots) were made quieter to make the important audio easier to hear. When important audio finishes, the noises whose volume was decreased are returned to their previous value.

Special audio effect such as gunshots and jumping, were implemented and already included with UFPS package. Ambient noises of the desert were included with environmental pack.

Extra background music was added to play while pause menu, end level menu and main menu were active to distinguish the non-playing state of the game with playing state.

3.4.1. Dialogue

Dialogue was used inside the tutorial level to teach the players how to play and lead them through the level. The “script” for the dialogue was written beforehand in a way to help convey a clear message with minimal amount of words [Figure 7]. After writing the desired dialogue, it needed to be voiced clearly by a native English speaker. However, voice actors do not work for free so an alternative approach was found.

Instead of using real people to voice the dialogue, virtual speech synthesis was used. There are many websites and companies offering speech synthesis, some of them even give users ability to try their engines online through their website. Two different speech synthesis providers were used to make the dialogue: NaturalReaders and LumenVox. NaturalReaders had a wide selection of English speaking voices that sounded pleasant and natural. NaturalReader voices Mike, Rich and Ryan were used to voice dialogue spoken by the player, his commander and his peers. LumenVox voice Rebecca was used to voice a help agent that tells the player how to use the application.

Commander: *That was quite a rough landing. Did you suffer any damage?*

Player: *I am a bit disoriented but otherwise fine.*

Commander: *Never mind the disorientation as long as your bullets can still hit their targets.*

Player: *I assure you, my shooting is not impaired.*

Commander: *Good! Prove it by shooting down all shooting targets in the area.*

Figure 7 - Example dialogue between commander and the player

3.5. Tutorial Scene

Tutorial scene was created by placing models of rock and boulders from environment asset pack around to shape the level. Cliffs were placed to box player in and ensure he has only one path to follow through the tutorial [Figure 8].

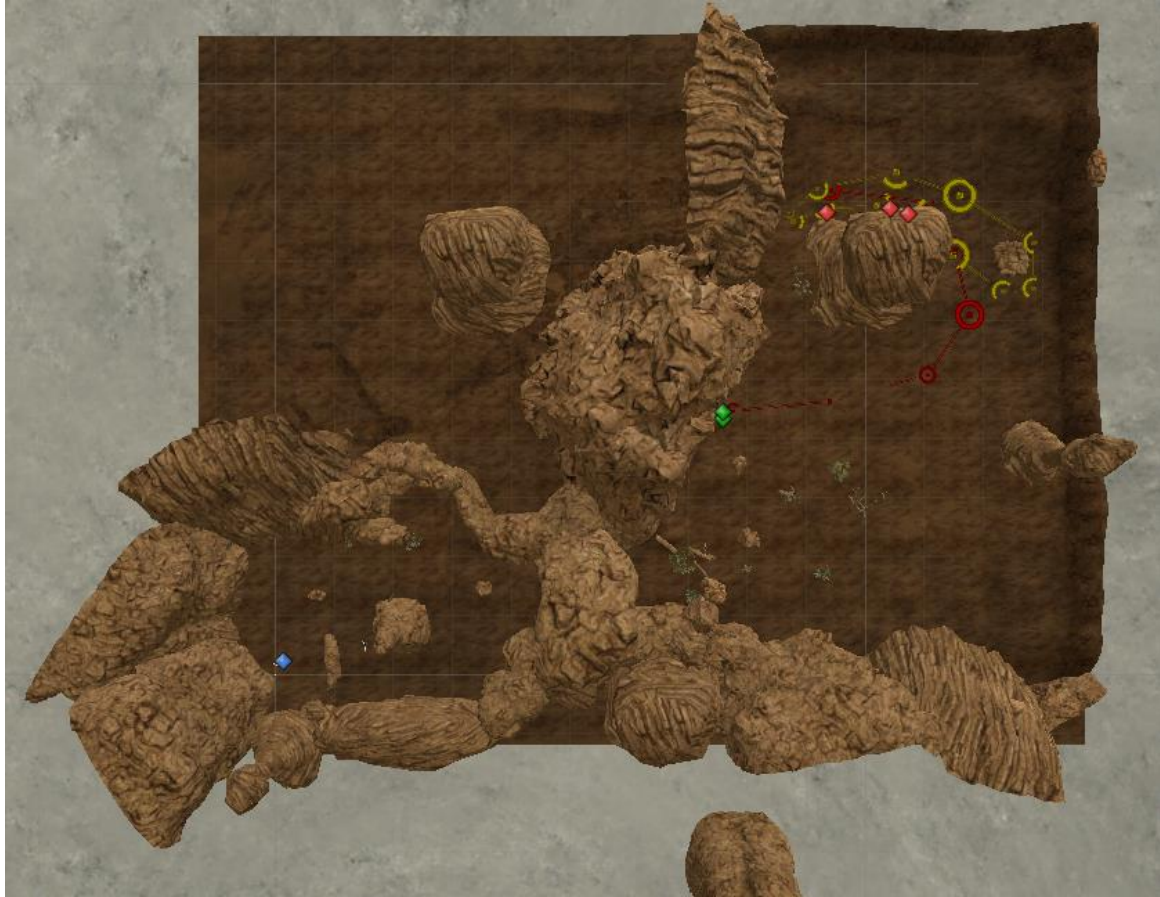


Figure 8 - Top view of the tutorial level. Blue marker denoting starting player position, green markers denoting friendly agents, red markers denoting enemy agents

Player starts the level surrounded by rock walls and is instructed to look around to find his commander. The commander is placed to the right of the player and rocks and grass were placed in between them [Figure 9].

Player is supposed to look around either using the mouse or by using his HMD. Player in game character orientation is always in the view direction, therefore, game checks if player character is facing the commander character. If that is the case, player is instructed to go towards the commander.



Figure 9 - Initial position of the commander

Since direct path from player's starting position and commander's position is blocked by small grass and rock, player is supposed to walk around the obstacles to reach the commander, effectively teaching him how to move in all directions.

When the player arrives in a set distance from the commander and is facing him, special sequence triggers in which commander is talking to the player. The commander then asks the player to shoot down all shooting targets that were placed in the area, in such a layout that the player can see them from his spot next to the commander.



Figure 10 - Shooting target

When player starts shooting, if he successfully hits targets, he is being praised by his commander and if he misses he is being scolded. That way the player gets feedback knowing whether he had indeed hit his target.

After shooting down 3 targets, the commander instructs the player how to shoot down the last target that is far away. The target is too far away to hit without using snipe functionality. Player is instructed how to activate the sniping mode to zoom in to the target and successfully shoot it.

When player succeeds, commander praises him and asks the player to follow him. The commander then proceeds to run further into the level until reaching a crumbling wall obstacle that prevents further progression. The purpose of the crumbling wall is to block the player from proceeding further until he reaches the point of the game in which the commander is asking him to shoot the wall. Player is then supposed to shoot into the crumbling wall enough times to make it crumble and break. The number of times the player needs to shoot the wall is adjusted so that the player ends up running out of ammo. This creates an excellent opportunity to teach the player how to reload.



Figure 11 - Crumbling wall that the player needs to shoot

After player reloads and finishes hitting the wall enough times, the wall crumbles and player proceeds to follow the commander. Commander runs further into the level and jumps over a fallen wall. When player reaches a set distance from the wall he is instructed to jump over the wall.

Upon doing so, the player is supposed to reach the commander again. This time the commander is waiting for him near several friendly agents. When the player reaches them, a new sequence is triggered. In this sequence commander talks to the agents and after the talk, instructs the player to clear the area of enemy units with the help of friendly units.

Player is supposed to use the learnt knowledge to search the area for enemy agents and dispose of them. Enemy and friendly agents have a basic shooter AI implemented using free RAIN state machine framework and is obtained as a part of a Shooter AI package. The integration of agent AI with UFPS and rest of the system was manually implemented through code modification and creation of new scripts.

AI agents walk around a set route until they spot their enemy (for enemy units allied units and the player are enemies) or until they get shot. When one of the enemy

agents gets shot or sees the player he notifies his allies so that all nearby agents start targeting the player.

Player needs to shoot down the enemy agents and avoid getting shot down himself. In case the player is shot enough times to drain his health, he is respawned at the starting location and needs to walk back to end of the level to finish disposing of enemy agents.

When player successfully disposes of all enemies in the level, the level ends and end level window is displayed notifying the player of his success and offering him a choice to exit the game or move on to the main level.

All logic of the tutorial level is implemented with simple triggers and actions.

Triggers are reusable behaviors whose purpose is to detect something happening and call specified methods when it happens. For example, `TriggerAfterAudioFinishes` is a behavior in which user specifies through Unity Editor which audio he is interested in and using Unity Events, specifies which external methods to call when that audio finishes. Because one can specify through the editor what one wants to happen, this trigger can make nearly anything happen as long as it has the reference to the behavior that implements the method that needs to be called.

Action behaviors are simple behaviors that implement methods that, previously mentioned, “trigger behaviors” call. Action behaviors don’t do anything by themselves.

Because of this trigger-action design, reusability of actions and triggers is increased. To put things into perspective, imagine if we want to make the game load another level if player touches a certain object. We would do that by writing a `LoadLevel` action in which we would tell which level to load and how to load it. We would then implement a trigger that checks for physics collision with an object, and set it up so that upon collision it calls methods of `LoadLevel` action.

If, later on, we decide we want the game to load the next level if something else happens, we would only implement a new trigger. And if later on we decide that we want player’s health to recover if he touches a specific object – we already have a

trigger that can notify actions of player touch, we would only need to implement action that would heal the player.

For commander object, which is supposed to have certain states, ICode state machine is used to keep track of the state [Figure 12].

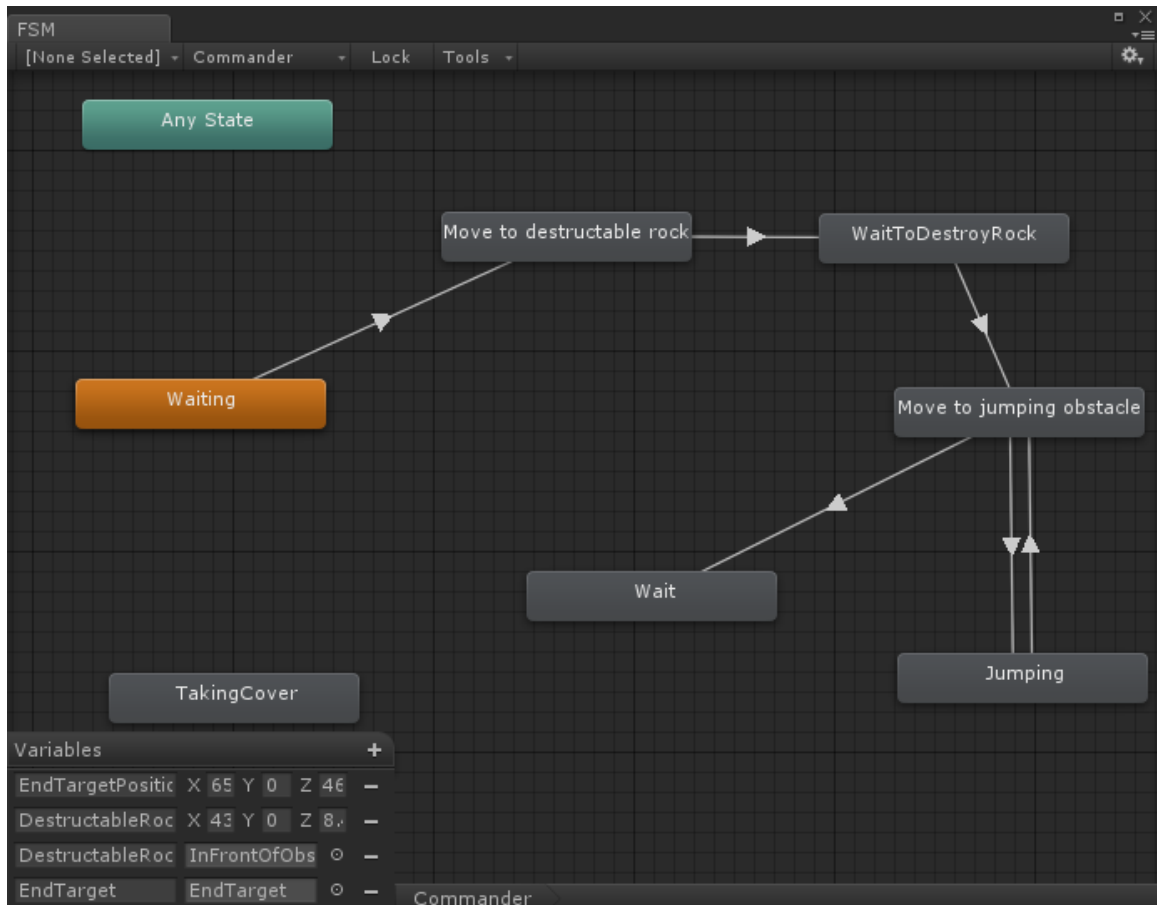


Figure 12 - ICode state machine for commander

3.6. Main Gameplay Scene

Main gameplay scene was based on a demo scene that was included with used desert environment pack to showcase it [Figure 13]. The included demo scene had an artistic quality to it and was a medium large open level scene like it was envisioned for main level to be. Because it corresponded to initial requirements so well, only minor changes were made – removing models that do not belong in Middle East, such as ancient statues and obelisks.

Besides implementation of randomizations and stressors that are better explained in following chapters, most of the work done on the main scene revolved around

AI behavior. A majority of behaviors and objects were already implemented in the tutorial scene and were just reused in the main scene.



Figure 13 - Main scene

RAIN package was used to create many waypoint routes that both friendly and enemy agents would follow in game. Once spawned, agents would stand in place, not moving until enemies are detected, however, that behavior is unrealistic so instead, agents have a patrol route assigned. Patrol routes were created manually (by hand) through RAIN interface.

When agents enter battle with hostile agents or player, their behavior is randomized. They can seek out enemies, or they can take cover. To take cover, special cover positions had to be created to notify agents where they will be able to shield themselves from enemy fire. The cover positions were also placed manually in appropriate places in the level.

3.6.1. Avoiding user learning by randomization

Repetitive play of the game might create unwanted learning effect in the users. Users might see the patterns and repetitiveness in the game and increase their performance by knowing what will happen next in the game.

This is an unwanted type of learning. Even though it increases the performance of users, it does not yield performance increase in real life scenarios. Not only that,

but from the performance data it is hard to tell if the performance increase is caused by genuine improvement in players' skills or them knowing what is going to happen next because they have already experienced it.

Naturally, this creates a problem. How can we rely on the performance data or be sure that the players are actually genuinely improving through playing the game?

To maximize the usefulness of the application we must minimize the unwanted learning effect as much as possible by minimizing repetitiveness of the game. If every play session differs from the last and players don't know what to expect, the unwanted learning effect would have minimal impact on the performance data and observed performance improvements would, ideally, translate into real life performance improvements.

There are many things players can get used to inside the game but the most important ones are: level layout, enemy behavior and positions of enemies.

If the level layout is the same, eventually, the player will start recognizing places and know where he is and where his enemies might be, where the cover positions are, etc. Best way to remedy this would be to generate random level layout, however this has not been implemented because random level generation is a big topic on its own and it is out of scope of this project.

Instead of generating the entire level randomly, it is also possible to spawn enemies in random positions and in random numbers. If the number of possible spawn points is large it becomes hard for players to remember and predict where enemies might be.

Also, instead of making the player start the game in the same position, the starting position can also be randomly determined to make it harder for the players to remember their surroundings from previous play sessions.

In the game 3 different randomizations have been implemented.

Random player spawn point

In the large level map 20 different player spawn locations were placed. Those locations are possible locations where player can start the game. Therefore, the player would need to play the game at least 20 times to see the level from all 20

different starting locations. On each player spawn point a *PlayerSpawnArea* script is placed along with a sphere collider that defines the radius of the spawn point [Figure 14].

At the start of the level, the first *PlayerSpawnArea* script that reaches code execution will call spawning of the player. It will collect the list of all available player spawn points and randomly pick one out of the list. The player will then be repositioned to start in that start point.

```
public class PlayerSpawnArea : MonoBehaviour {

    public static PlayerSpawnArea[] Areas;
    public static PlayerSpawnArea lastSpawnArea;

    void Awake()
    {
        //if areas is empty, fill it
        if(Areas==null)
        {
            Areas =
GameObject.FindObjectsOfType<PlayerSpawnArea>();

            //pick a random spawn area
            var index = Random.Range(0, Areas.Length);
            Areas[index].Spawn();
        }
    }
    public void Spawn()
    {
        //get a random point within a circle
        var offset = Random.insideUnitCircle;
        var collider = GetComponent<SphereCollider>();
        var finalDestination = new Vector3(
            transform.position.x + offset.x*collider.radius,
            transform.position.y,
            transform.position.z + offset.y * collider.radius
        );
        //get player
        var player = GameObject.FindWithTag("Player");
        //move player
        player.transform.position = finalDestination;
        //apply the rotation of this object
        player.transform.rotation = transform.rotation;
    }
}
```

Figure 14 - PlayerSpawnArea script

Randomized agent count and spawn locations

In addition to random player spawn locations, ally and enemy agents also have random spawn locations. However, ally and enemy spawn locations also have a range of agents each of the locations might spawn. Therefore, even if a certain location is chosen, it is impossible to tell whether that location will have one agent in it or multiple agents.

Agent spawn points were placed all over the level totaling to 35 ally agent spawn points and 42 enemy spawn points [Figure 17].

Similarly to how Player spawning works, agent spawn points are also marked by an object with a sphere collider and *NpcSpawnArea* script [Figure 15]. However, unlike how *PlayerSpawnArea* works, *NpcSpawnArea* does not initiate spawning of agents by itself. Instead, spawning of agents is initiated by a *SpawnController* script placed on a parent object that contains all *NpcSpawnAreas* of the same type (type being either ally or enemy) [Figure 16].

```
public class NpcSpawnArea : MonoBehaviour
{
    private static WaypointRig[] Routes;
    public int MaxNumberOfSpawnedObjects = 3;
    public void Spawn(GameObject prefab, GameObject parent)
    {
        var spawned = Instantiate(prefab);
        //get a random point within a circle
        var offset = Random.insideUnitCircle;
        var collider = GetComponent<SphereCollider>();
        var finalDestination = new Vector3(transform.position.x +
offset.x * collider.radius,transform.position.y,
        transform.position.z + offset.y * collider.radius);
        //move spawned
        spawned.transform.position = finalDestination;
        //random rotation
        spawned.transform.Rotate(Vector3.up, Random.Range(0, 360));
        //reparent
        spawned.transform.parent = parent.transform;
        //set up patrol behaviour
        if (Routes == null)
            Routes = GameObject.FindObjectsOfType<WaypointRig>();
        //pick route
        var route = Routes[Random.Range(0, Routes.Length)];
        // set route
        var ai = spawned.GetComponentInChildren<AIRig>().AI;
        ai.WorkingMemory.SetItem<string>("patrolRoute",
route.name); }}
```

Figure 15 - NpcSpawnArea script

This script collects all of its children *NpcSpawnAreas* into a list which is then shuffled with Fisher-Yates shuffling algorithm [4]:

```
To shuffle an array a of n elements (indices 0..n-1):  
  for i from n - 1 downto 1 do  
    j ← random integer such that 0 ≤ j ≤ i  
    exchange a[j] and a[i]
```

The script then goes from the beginning of the list in order and assigns a random number of agents from given (min, max) interval to each spawn point until the max number of agents per level is reached.

```
public class SpawnController : MonoBehaviour {  
  
    public GameObject PrefabToSpawn;  
    public GameObject HolderParent;  
    public int NumberToSpawn = 40;  
  
    private NpcSpawnArea[] Areas;  
  
    void Awake () {  
        // get all areas  
        Areas = GetComponentsInChildren<NpcSpawnArea>();  
        Areas.Shuffle();  
  
        for(int spawnedCount=0, areaIndex=0; spawnedCount  
<NumberToSpawn; areaIndex++)  
        {  
            var area = Areas[areaIndex];  
            var numberToSpawn = Random.Range(0,  
area.MaxNumberOfSpawnedObjects+1);  
            if(numberToSpawn>NumberToSpawn-spawnedCount)  
                numberToSpawn = NumberToSpawn-spawnedCount;  
  
            for(int i=0; i< numberToSpawn; i++)  
            {  
                area.Spawn(PrefabToSpawn, HolderParent);  
                spawnedCount++;  
            }  
        }  
    }  
}
```

Figure 16 – SpawnController script

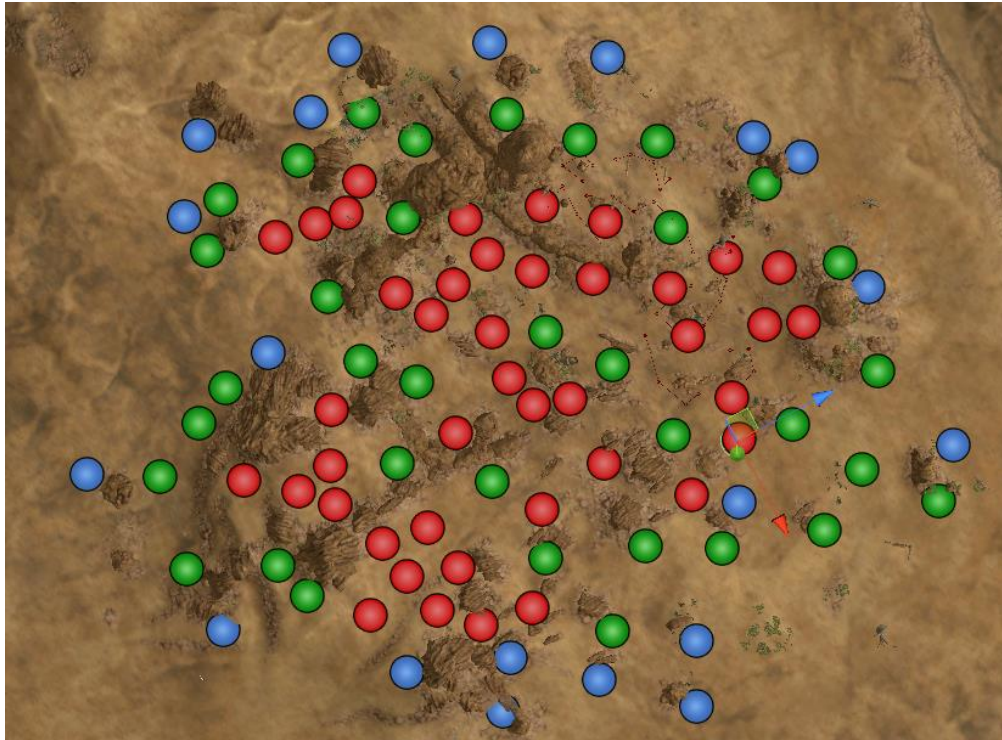


Figure 17: Map of the level with spawn points shown, Blue circles = player spawn points, Green circles = ally spawn locations, Red circles = enemy spawn locations

3.7. Integration with Oculus Rift

Oculus Rift is a virtual reality head mounted display (HMD) developed by Oculus VR. Oculus Rift provides a way to see inside the game that gives a sense of presence inside the virtual environment that cannot be achieved with standard monitors. Seeing a slightly different image in each of the eyes gives unparalleled sense of 3 dimensional world and distance between objects, immersing the players even more into the world. Besides providing the display of virtual world, Oculus Rift also provides head tracking that can detect the position and rotation of the head which can then be translated into the camera movements inside the game.

Since the aim of this project is to make the game as close to reality as we can, usage of a HMD is integral to achieving that. Since Oculus Rift is still in active development and a consumer version has not been released, the device used in the making of this project is Oculus Rift DK2 (Development Kit 2) [Figure 18].

Oculus Rift DK2 has a resolution of 960 x 1080 pixels per each eye and 75 Hz refresh rate. [5]



Figure 18 - Oculus Rift Development Kit 2

Enabling usage of Oculus Rift was done directly through native Unity 5.1 support for the device. This only required the installation of Oculus Rift Runtime and checking a “Virtual Reality Supported” checkbox in Unity Player settings.

Upon running the built application, the application built with Unity 5.1 checks if Oculus Rift is available and if it is changes its rendering to support it. This native support does a few optimizations behind the scenes that were previously impossible by using a plugin. This in turn makes the game run faster.

This simplifies the workflow significantly because everything remains the same as if we were not using Oculus Rift. We still have one camera through which we see the world – however, that one camera is automatically rendered in stereoscopic vision required for Oculus Rift. Also, head rotation is translated automatically to said game camera.

All that was left to do in the project was read that change in camera rotation and move the rest of the player’s body to follow it. This was done by attaching a *SyncCameraRotationToPlayer* script onto the Player object which synchronizes the camera rotation with player body rotation [Figure 19].

```

public class SyncCameraRotationToPlayer : MonoBehaviour {

    public GameObject Camera;
    public GameObject Player;

    vp_FPPlayerEventHandler FPPlayer;

    public void LateUpdate()
    {
        // remember current rotation of the player
        var currentRot = Player.transform.rotation.eulerAngles;
        // remember current camera rotation in the Y axis
        var newY = Camera.transform.rotation.eulerAngles.y;
        // rotate the player to the current camera Y axis rotation

        if (FPPlayer == null)
            FPPlayer = Player.GetComponent<vp_FPPlayerEventHandler>();
        FPPlayer.Rotation.Set(new Vector3(currentRot.x, newY));

        // because camera is the child of player, rotating the player will
        rotate the camera as well.
        // Therefore we have to reset the camera rotation to what it was
        before this change.
        currentRot = Camera.transform.rotation.eulerAngles;
        Camera.transform.rotation.SetEulerAngles(currentRot.x, newY,
currentRot.z);
    }
}

```

Figure 19 - SyncCameraRotationToPlayer script

However, this set up does not decouple mouse movement from head movement which means that the camera will rotate along with player's head movement but can also be rotated by moving the mouse. This behavior is not satisfactory because we do not want the mouse movement to rotate our game view. The fix to this issue is closely related to Delta Six support and is explained in the next chapter.

3.8. Integration with Delta Six

The VR application developed as a part of this project needs to be controlled intuitively and easily while the user is wearing a HMD, which means that standard input schemes such as usage of mouse, keyboard or traditional gamepads is not adequate. The player in the game needs to be able to aim and shoot inside the game as realistically as possible because the purpose of the application is to help improve soldier's response to stress in combat – and soldiers do not go into combat with a gamepad. This creates a need for an alternate controller that would feel as close to the real guns as possible.

This is where Delta Six comes in to solve that issue. Delta Six is a wireless gun controller – a controller in shape of a gun, made by Avenger Advantage [Figure 20]. The user of the Delta Six uses it as if it were a real gun – he aims and shoots with it like it was the real thing.

Delta Six has several sensors inside that translate the movement of the controller into movement of the mouse and sends it to its Bluetooth receiver. The Bluetooth receiver pretends to be a combination of a mouse and keyboard when plugged in to the PC. The vertical and horizontal rotation of the Delta Six is translated into mouse movement, while key presses are translated into keyboard key presses.



Figure 20 - Delta Six Controller

Because of this behavior integration with Delta Six did not require integration with a specific SDK or extern library. The application just needed to support mouse and keyboard control and by extension, it would support Delta Six controller.

However, because 360 degree turns of a controller need to translate into 360 turns inside the game, mouse sensitivity needed to be adjusted to just a right amount for the movements to line up.

It is also important to take into account that even if mouse sensitivity is perfectly adjusted, sooner or later, because of the angle drift caused by lack of precision the controller will get misaligned to its virtual representation. In such scenario it is important to recalibrate the controller and realign it to its virtual representation. To

achieve that, a software calibration was implemented that activates while holding R3 button on the controller. While user is holding that button the game freezes its virtual gun in place which allows the user to reposition his controller into place. Upon releasing the button, control of the game returns to the player and he can again control his aiming with Delta Six.

Besides movement, the game has a few more possible actions the player can perform such as shoot and move. Inside the game movement of the player is being controlled by an analogue stick on the controller, while all other actions are mapped to the corresponding buttons on the controller. You can find the specific button mapping information in the Software Usage Instructions chapter.

Lastly, the only thing left to do was decoupling of mouse movement from camera movement. Players will be wearing Oculus Rift HMD which allows them to freely look around the game. This causes the camera to rotate around to mimic their real life head rotation. Initially, the shooting direction inside the game always corresponded to the center of the screen, and moving the mouse would rotate that camera as well. This approach is not good for VR because we do not want the players to “aim with their head” nor do we want their gun to control their head movement. We have a dedicated controller for the aiming – Delta Six and dedicated HMD for rotating the head. Therefore, the movement of the gun should not impact the movement of the camera and vice versa. This would allow the player to e.g. look to the left while shooting forward, or look at what the enemy is holding in their hands while shooting towards their head.

This required extensive changes to the system by introducing a new variable – shooting direction and modifying gun controllers to shoot in that new direction. It also required shifting the cross hair representation that shows players where they are aiming to the new position.

3.9. Increasing Graphical Fidelity

Graphical fidelity is important for a success of a game. Even more so for a game centered on a realistic VR experience. Poor graphics will not let the players forget that they are playing a game. We must make their experience close enough to the real life so that the experience of playing a game would come as close to fighting

a real world battle as possible. Lesser the difference between real world and virtual one, more effective the training can prove to be in real life situations.

To achieve the desired level of realism, a high quality models and textures were used for the design of the levels in the game. Shaders for all in game objects use a physically based shading model that approximates how light behaves in the real world, giving object realistic appearance.

Physically based shader that comes with Unity 5 is used in conjunction with baked global illumination, high dynamic range (HDR) rendering and ambient occlusion for even more realistic look.

Relief Terrain Pack was used for shading of the terrain. The pack has a custom terrain shader included that has height based texture blending, offsetting textures to minimize tiling effect and parallax mapping which make the flat terrain seem non flat [Figure 21].

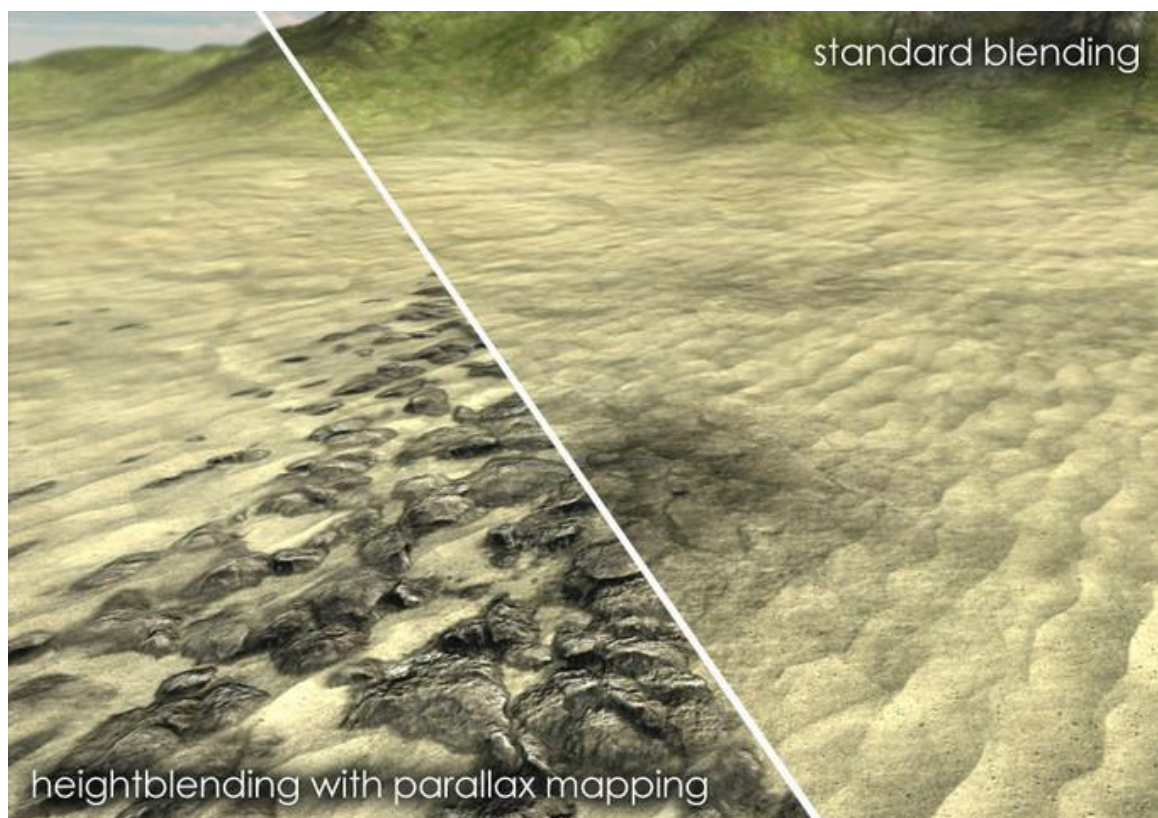


Figure 21 - custom shader with height blending and parallax mapping vs standard Unity terrain shader

However, even the high quality shaders used could not fix the red tint of the scene caused by texture and light colors which were uncharacteristic for Middle East

deserts [Figure 22]. To make the available models look closer to Afghanistan desert, they were changed in GIMP - free image manipulation application.



Figure 22 - appearance of the level before image manipulation

A seamless sand texture was used to change existing reddish textures to a more yellowish tone [Figure 23]. The sand texture was placed over red textures and used in Color blending mode to change the underlying colors into a different tone.



Figure 23 - used seamless sand texture

After changing the textures, ambient light color and directional light colors were changed into a light yellow, almost white color [Figure 24].



Figure 24 - appearance of the level after texture and light changes into a more yellowish tone

As a finishing touch, a few post processing effects were added. A bloom effect was added to give a bit of a glow to the scene, and to cause bleeding of color when the player looks into overly bright areas.

After that an antialiasing FXAA (fast approximate anti-aliasing) effect was added to make the edges of geometry smooth which completed the realistic look of the scene.

3.10. Performance Improvements

Higher the graphical fidelity of the game, more expensive it becomes to render. VR applications make it even harder for the GPU (graphics processing unit) to render the scene. If it takes the GPU too much time to render the scene the game will seem unresponsive and “laggy” and therefore unplayable.

To avoid those issues we need to perform as many performance optimizations as possible and sometimes sacrifice graphical quality for performance.

In this project, many graphical performance improvements were implemented to make the game framerate never fall below 75 fps (frames per second). 75 Hz is the refresh rate of Oculus Rift and therefore it was set as a goal minimum fps.

Implemented performance improvements include:

Optimized Post Processing effects

Post processing effects are expensive and cause a relatively large performance hit when used. To minimize that performance hit while still retaining increased graphical fidelity from their usage, their parameters were heavily optimized to cause minimal performance hit. Where available, “mobile” version of effects were used – instead of using the usual Bloom post processing effect, the Optimized Bloom was used instead.

When choosing an antialiasing implementation, FXAA was chosen as it provided the best antialiasing effect with minimal performance impact.

Baked shadows

Highest impact on the performance comes from calculation of shadows and light. Therefore all static objects had their shadows and ambient occlusion baked at edit time. Dynamic objects that could not be baked had their shadows calculated at runtime, however, since there were limited number of moving objects in the scene it didn't cause a performance hit.

Occlusion culling

Occlusion culling is a technique that disables rendering of objects when they are not currently seen by the camera because they are obscured by other objects. Unity's built in occlusion culling was used to determine visible geometry at runtime and lower the amount of drawn objects therefore increasing performance.

Fog & Camera clipping planes

Most performance issues from open space scenes in FPS games come from the ability to see far to the horizon. Increasing view distance and view volume of the camera increases number of objects that need to be drawn. The opposite is true as well, so to increase performance we need to decrease camera view volume. View volume is controlled by camera near and far clipping planes, or rather the distance between them.

Decreasing the distance between them increases the performance, however, if the far plane is set too close to the near plane the objects will start visibly disappearing from view.

To benefit from lowered view volume we need to make the objects not visibly disappear. This is where fog comes in, fog blends the color of the generated pixels down towards a constant color based on distance from camera. It makes the objects gradually disappear into the distance [Figure 25].

All that is then left to do is adjust camera clipping to occur behind the fog, where it won't be visible to the player.



Figure 25 - Fog effect

Levels of Detail (LOD)

When an object in the scene is a long way from the camera, the amount of detail that can be seen on it is greatly reduced. However, the same amount of triangles will be used to render the object even though the detail won't be noticed.

Level of Detail (LOD) rendering reduces the number of triangles rendered for an object as its distance from camera increases.

Most objects used in the creation of the project have 3 LOD levels set up to increase performance.

Player settings

In the player settings section of the editor there are many things that can be checked to improve performance of the built game, such as activating static and

dynamic batching of geometry, using GPU skinning, using DX11 instead of DX9, prebaking collision meshes and optimizing mesh data.

A test was done to see whether deferred or forward rendering would be faster along with testing of three different DirectX versions. The result showed that forward rendering was faster – as expected, since the scene only contains 2 directional lights (deferred rendering works better with more lights). The result also showed that higher versions of DirectX yield better performance, which is also not surprising since DirectX 11 and DirectX 12 contain new rendering optimizations.

Therefore, DirectX 12 was used in combination with forward rendering.

Automatic performance adjustment

When you cannot increase performance without sacrificing graphical fidelity then an important question arises: what to sacrifice and when?

Low end hardware might need a lot of sacrifice in graphical fidelity to increase performance but better hardware might not need tradeoffs at all.

To ensure the players have the best possible graphics while maintaining their target framerate a special system was implemented.

The system contains two major components: a FPS counter and automatic performance manager. FPS counter is active during the game and constantly measures game FPS, averaging it over a certain number of frames. The counter is told what the target framerate is and what framerate is considered to be “too high”. When the framerate becomes too low or too high the counter notifies another script doing the automatic quality adjustment.

Automatic performance manager has a list of possible performance vs quality tradeoffs. Such performance tradeoffs include switching high end effects for low end effects or turning them off. Automatic quality management script then goes through the list and turns features on/off to increase performance if the framerate is too low, or to increase graphics if performance is good enough.

This runtime system makes it easier for developers because it ensures that the user will always have the perfect performance vs graphics ratio for his machine. Even if his machine suddenly becomes slower or faster, because of background

processes and other applications running simultaneously on the system, or if he enters a more performance heavy scene than the previous one and vice versa.

Agent AI management

By increasing the number of AI agents in the scene a noticeable CPU performance hit was noticed. The performance hit was so large that the application became CPU bound instead of previously GPU bound.

Each AI is expensive because it needs to make decisions and continuously raycast to determine what is in its field of vision.

To remove the performance hit there were two options that could be taken. Either lower the number of AI agents in the scene until the application again becomes GPU bound, or lower the number of *active* agents in the scene.

The second approach was taken and each AI received *DistanceFromPlayerTrigger* script that activated/deactivated the agent AI when the agent entered/exited a certain radius from the player. That way only a limited number of agents is active at the same time. This decreases the performance hit while still allowing the scene to contain a large amount of agents.

4. Stressors

It is a known fact that emotional stressors may have harmful effects on human cognitive-motor performance. Soldiers in war are subject to many unpredictable disruptions such as sudden shootouts, bombings, seeing their colleagues getting hurt, loud noises, etc. Resistance to stressors may well be the difference between life and death. That is why it is important to try to help individual soldiers increase their resistance to emotional stressors to improve their battle performance in real life. The developed project contains features that allow setting up of stressors and measuring their impact on user performance.

Currently there are two different stressor types implemented but there is room to implement other types of stressors in the future such as airblast system, sudden noises, etc.

The stressors are triggered after a certain game time and last only a short moment. In that moment the game is briefly paused so that agents do not shoot the player while he is unable to do anything about it. After stressor deactivates the game is unpaused and player can continue the game.

The game has an option to turn stressors on or off. That way, performance without stressors can be measured to compare it to the performance with appearance of stressors.

4.1. Image Interruptions

Full screen images are implemented as the part of Unity uGUI system. They are made out of an Image component and *AspectRatioFitter* component that stretches the image to fit the screen [Figure 26].



Figure 26 - image used as a stressor

4.2. Video Interruptions

Video interruptions are almost identically implemented as Image interruptions. The difference being that video inside Unity is imported as a video texture.

To import video inside Unity, Apple QuickTime software is needed. After installing the software, the PC needs to be restarted otherwise Unity will be unable to detect the plugin and won't import the video.

A free .avi video file of an explosion was used as a stressor [Figure 27].



Figure 27 - Video stressor

5. User Performance Measurement

HitTracking behavior is responsible for tracking gun fire. It tracks total shots fired, how many of those shots have hit friendly targets and how many have hit enemy targets. The rest of the hits were actually misses. From that information shot accuracy is calculated as: $1 - \text{MissedShots} / \text{TotalShots}$.

The same script also keeps track of how many times the player has been hit by enemy fire, how many times he “died” in game, how many enemies player managed to take down and the total number of enemies in the level.

Along with hit data, the script also logs times when specific stressors appeared.

Whenever one of the variables the script is tracking changes value, the script saves the timestamp and the new variable value to a list which is later on saved to a file that can then be opened with external applications for viewing.

When level gets completed the list of timestamps and values are written into a file in the same directory as the built application.

Many other scripts provide data to *HitTracking* script. Each hittable target has a specific script that detects shots and determines who shot who so the appropriate value can be raised in the *HitTracking* script. Each damageable entity also has a script that notifies *HitTracking* of deaths.

The project does not currently include a way to graphically view the performance data. However, there are many software applications out there that can help visualize exported data (such as Microsoft Excel).

6. Future Works

The system developed as a part of this work can be extended and improved in various ways to make a more complete solution:

Dedicated application for viewing performance data

Developed application measures various performance data, however, there is no dedicated application that can read that data and turn it into graphs and charts that would help us better understand that data. Making of such application would be the next step in the further development.

Integration with VR treadmill devices

As of making of this work, there exist a few VR devices whose usage would enhance the realism and game experience by providing a natural movement inside the virtual environment. Those devices are Cyberith Virtualizer [6][Figure 28] and Virtuix Omni [7] VR treadmills. However, both of the devices are still in active development and are not yet publicly available. In the future, when they do become available, it would be a good idea to add their native support into the application.



Figure 28 - Cyberith Virtualizer in action

Increased graphical realism and fidelity

Virtual Reality requires good PC specifications to reap its full benefits, however, such hardware was unavailable while developing this application. This caused many tradeoffs of graphical fidelity for the performance benefit. Access to the better PC hardware would open doors to further increases of graphical fidelity and realism of the application.

Adding more stressor types

In this work only basic video and image stressors were implemented, however, there are many more possible types of stressors that could be added in the future such as airblast and acousto-haptic vibrations.

Developing a smart AI

To make the system more valuable, challenges the players must face should be made more formidable by creating smarter opponents. However, creating such a good AI was out of scope for this project, therefore it has not been done.

Random level generation

Another addition that would make the system better would be further minimizing the effect of user learning by adding more randomization such as random level

generation. Taking the randomization to the next step by creating randomized levels could provide the biggest benefit in minimizing unwanted user learning, however, developing such a feature is complicated, time consuming and out of scope of this project.

7. Conclusion

Serious games have a great potential to help us in our work. As shown by this project, potential for VR applications to bring us into a realistic environment is only limited to our access to needed hardware. With Oculus Rift HMD and Delta Six gun controller in combination with realistic beautiful graphics that still have high performance on mid-tier hardware, it is possible to bring the immersion to the next level. Making the game intuitive and educational enough to teach the players how to play through the game itself is also possible and has been achieved, proving that games have great educational potential that should be further explored.

The base gameplay of the first person shooter is implemented but needs further work and polish to turn into a completely engaging experience. Still, the application is easy to use and serves its intended purpose of collecting performance data, with and without basic stressor interruptions, quite well.

A lot of work has been put into making of this project, however, there is even more left to do to turn it into a complete solution to help soldiers in their training such as adding support for VR treadmills, making dedicated application to read performance data, development of a smarter AI and increase of the graphical realism of the environment.

Ana Stepić:

8. Sažetak

Sustav za praćenje utjecaja stresnog okruženja računalne igre na kognitivne sposobnosti čovjeka

U radu je opisan postupak izrade igre pucanja iz prvog lica kao ozbiljne igre unutar Unity okruženja te njena integracija s dostupnim VR uređajima: Oculus Rift i Delta Six. Razmatraju se rješenja problema koji se javljaju pri izradi realističnih VR aplikacija, poput potrebe za optimizacijom performansi i poboljšanje grafičke privlačnosti, te kako dizajnirati ozbiljnu igru kako bi pružala što bolje iskustvo igračima te ih educirala na intuitivan način.

Svrha dotične igre je mjerenje utjecaja stresnih čimbenika: slikovnih i video prekida, na kognitivne performanse igrača u svrhu pomoći pri poboljšanju njihovih performansi u stvarnim životnim situacijama. Kako bi se dobili što relevantniji podatci o performansama razmatraju se i tehnike kako smanjiti nepoželjni efekt učenja igrača koji ponavljaju test više puta.

Ključne riječi: *Stresori, Stres test, Kognitivne performanse, mjerenje performansi, Unity, Oculus Rift, Delta Six, Virtualna stvarnost (VR), 2D PC igra, ozbiljna igra, pucačina iz prvog lica*

9. Abstract

System for monitoring the impact of stressful computer game environment on human cognitive capabilities

This thesis contains methods for making a 3D first person shooter (FPS) serious game inside Unity engine and its integration with available VR devices: Oculus Rift and Delta Six. Different solutions to problems appearing in development of VR application are considered, such as need to optimize performance and increasing graphical fidelity. How to design a serious game to offer as good of an experience as possible and educate the players in an intuitive way has also been considered.

The purpose of the developed game is tracking human cognitive performance when subjected to image and video stressors for the purpose of helping the players improve their performance in stressful real life situations. To make collected data as relevant as possible, different techniques are considered that can help reduce unwanted learning effect present in users repeating the same test multiple times.

Keywords: *Stressors, Stress test, Cognitive Performance, Unity, Oculus Rift, Delta Six, Virtual Reality (VR), 3D PC Game, Serious Game, First person shooter*

10. References

- [1] Statista, "Global Video Games Revenue," [Online]. Available: <http://www.statista.com/statistics/237187/global-video-games-revenue/>.
- [2] Venturebeat, "Global game revenues in 2014," [Online]. Available: <http://venturebeat.com/2014/06/24/gamer-globe-the-top-100-countries-by-2014-game-revenue/>.
- [3] Statista, "Global Box Office Revenue," [Online]. Available: <http://www.statista.com/statistics/271856/global-box-office-revenue/>.
- [4] "Fisher-Yates shuffle," [Online]. Available: http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle. [Accessed 2015].
- [5] RoadToVR, "Oculus Rift DK2 Info," [Online]. Available: <http://www.roadtovr.com/oculus-rift-developer-kit-2-dk2-pre-order-release-date-specs-gdc-2014/>.
- [6] Cyberith, "Cyberith Virtualizer Homepage," [Online]. Available: <https://cyberith.com/>.
- [7] Virtuix, "Virtuix Omni Homepage," [Online]. Available: <http://www.virtuix.com/>.
- [8] Unity Technologies, "Unity Manual," [Online]. Available: <http://docs.unity3d.com/Manual/index.html>. [Accessed 2015].
- [9] Unity Technologies, "Unity Script Reference," [Online]. Available: <http://docs.unity3d.com/ScriptReference/index.html>. [Accessed 2015].
- [10] VisionPunk, "VisionPunk (UFPS) Forum," [Online]. Available: <http://visionpunk.vanillaforums.com/>. [Accessed 2015].
- [11] VisionPunk, "UFPS Manual," [Online]. [Accessed 2015].

- [12] "Natural Readers Home Page," [Online]. Available: <http://naturalreaders.com/>. [Accessed 2015].
- [13] "LumenVox Text to Speech," [Online]. Available: <http://www.lumenvox.com/products/tts/#>.
- [14] "Master Audio Documentation," [Online]. Available: https://dl.dropboxusercontent.com/u/40293802/DarkTonic/MA_ReadMe.pdf. [Accessed 2015].
- [15] Unity Technologies, "Unity Asset Store," [Online]. Available: <https://www.assetstore.unity3d.com>. [Accessed 2015].
- [16] "Lean Tween Documentation," [Online]. Available: <http://dentedpixel.com/LeanTweenDocumentation/classes/LeanTween.html>. [Accessed 2015].
- [17] "Unity Wiki - FramesPerSecond," [Online]. Available: <http://wiki.unity3d.com/index.php?title=FramesPerSecond>. [Accessed 2015].
- [18] "Full Inspector Documentation," [Online]. Available: http://jacobdufault.github.io/fullinspector/guide/#docs/intro_dlls. [Accessed 2015].
- [19] "Math behind vision cones in Unity," [Online]. Available: <http://gameconn.blogspot.com/2013/01/the-math-behind-vision-cones-in-unity-3d.html>. [Accessed 2015].
- [20] Unity Technologies, "Unity 5 Audio Tutorials," [Online]. Available: <http://unity3d.com/learn/tutorials/modules/beginner/audio>. [Accessed 2015].

List of Figures

FIGURE 1 - GLOBAL GAME REVENUES IN 2014	5
FIGURE 2 - USAGE OF SERIOUS GAMES IN VARIOUS FIELDS AND INDUSTRIES	6
FIGURE 3 - DASHTI MARGO DESERT IN AFGHANISTAN	7
FIGURE 4 - POP UP WINDOW TEACHING THE PLAYER ABOUT GAME PLAY	12
FIGURE 5 - PAUSE WINDOW	12
FIGURE 6 - AUDIO MIXER	13
FIGURE 7 - EXAMPLE DIALOGUE BETWEEN COMMANDER AND THE PLAYER	14
FIGURE 8 - TOP VIEW OF THE TUTORIAL LEVEL. BLUE MARKER DENOTING STARTING PLAYER POSITION, GREEN MARKERS DENOTING FRIENDLY AGENTS, RED MARKERS DENOTING ENEMY AGENTS	15
FIGURE 9 - INITIAL POSITION OF THE COMMANDER	16
FIGURE 10 - SHOOTING TARGET	17
FIGURE 11 - CRUMBLING WALL THAT THE PLAYER NEEDS TO SHOOT	18
FIGURE 12 - ICODE STATE MACHINE FOR COMMANDER	20
FIGURE 13 - MAIN SCENE	21
FIGURE 14 - PLAYERSPAWNAREA SCRIPT	23
FIGURE 15 - NPCSPAWNAREA SCRIPT	24
FIGURE 16 – SPAWNCONTROLLER	25
FIGURE 17: MAP OF THE LEVEL WITH SPAWN POINTS SHOWN, BLUE CIRCLES = PLAYER SPAWN POINTS, GREEN CIRCLES = ALLY SPAWN LOCATIONS, RED CIRCLES = ENEMY SPAWN LOCATIONS	26
FIGURE 18 - OCULUS RIFT DEVELOPMENT KIT 2	27
FIGURE 19 - SYNCCAMERAROTATIONTOPLAYER SCRIPT	28
FIGURE 20 - DELTA SIX CONTROLLER	29
FIGURE 21 - CUSTOM SHADER WITH HEIGHT BLENDING AND PARALLAX MAPPING VS STANDARD UNITY TERRAIN SHADER	31
FIGURE 22 - APPEARANCE OF THE LEVEL BEFORE IMAGE MANIPULATION	32
FIGURE 23 - USED SEAMLESS SAND TEXTURE	32
FIGURE 24 - APPEARANCE OF THE LEVEL AFTER TEXTURE AND LIGHT CHANGES INTO A MORE YELLOWISH TONE	33
FIGURE 25 - FOG EFFECT	35
FIGURE 26 - IMAGE USED AS A STRESSOR	38
FIGURE 27 - VIDEO STRESSOR	39
FIGURE 28 - CYBERITH VIRTUALIZER IN ACTION	41
FIGURE 29 - DELTA SIX BUTTON MAPPINGS	54
FIGURE 30 – MAIN GAMEPLAY BUTTONS ON DELTA SIX	55
FIGURE 31 - ANALOGUE STICK AND START BUTTON ON DELTA SIX	55

11. Abbreviations

3D	3 dimensional
CPU	Central processing unit
DK2	Development Kit 2
FPS	First person shooter
fps	Frames per second
FXAA	Fast approximate anti-aliasing
GPU	Graphics processing unit
GUI	Graphical user interface
HDR	High Dynamic Range
HMD	Head mounted display
LOD	Levels of detail
PTSD	Post-traumatic stress disorder
uGUI	Unity new GUI. GUI implementation that was introduced in Unity 4.6.
UI	User interface
VR	Virtual Reality

12. Appendix

12.1. List of Used Third Party Assets

All resources used are used with permission in regards to their specific license.

Big Environment Pack 3	https://www.assetstore.unity3d.com/en/#!/content/13198 Environment package containing used desert models, textures and plants.
-------------------------------	--

Explosion Video	http://www.videezy.com/elements-and-effects/614-explosion Free video of an explosion used as a video stressor inside the application.
------------------------	--

Fire image	http://www.stockvault.net/photo/118828/fire Free stock image of a fire used as a static image stressor
-------------------	---

Full Inspector	https://www.assetstore.unity3d.com/en/#!/content/14913 Provides customization of Unity inspector and adds support for serialization/deserialization of generic types that are not natively supported in Unity
-----------------------	---

I2 Localization	https://www.assetstore.unity3d.com/en/#!/content/14884 Provides assets and scripts needed to have a localization functionality in Unity. This allows for exporting and exporting text strings for easy editing and fixing mistakes without having to change anything inside the actual game.
------------------------	--

ICode	https://www.assetstore.unity3d.com/en/#!/content/13761 Package that implements and offers simple state machine functionality
--------------	--

Imphenzia Soundtrack Music Loops	- https://www.assetstore.unity3d.com/en/?#!/content/3863 Package containing music loops that were used to provide music playback during menus.
---	--

Lean Tween	https://www.assetstore.unity3d.com/en/?#!/content/3595 Tween engine used for smooth UI transitions
Master Audio	https://www.assetstore.unity3d.com/en/?#!/content/5607 Asset package that provides simplified and more feature rich usage of Unity audio. It allows for audio pooling and randomized audio picking.
Prefab Evolution	https://www.assetstore.unity3d.com/en/?#!/content/17557 Plugin which allows prefab nesting.
RAIN AI	https://www.assetstore.unity3d.com/en/?#!/content/23569 State machine based AI system.
Relief Terrain	https://www.assetstore.unity3d.com/en/?#!/content/5664 Package providing AAA quality terrain shaders
Rewired	https://www.assetstore.unity3d.com/en/?#!/content/21676 Advanced input system that simplifies dealing with input and allows usage of third party controllers.
Sand Texture	http://wallpoper.com/images/00/30/67/53/textures-soil_00306753.jpg Used to change reddish hue of the original level textures into a more yellowish tone
Squad Command	https://www.assetstore.unity3d.com/en/?#!/content/23526 Extension for RAIN engine that provides shooter specific AI behaviors. Used for AI units inside the application.
SSAO Pro	https://www.assetstore.unity3d.com/en/?#!/content/22369 Screen space ambient occlusion post processing effect used to increase graphical fidelity.
UFPS	https://www.assetstore.unity3d.com/en/?#!/content/2943 Template for making FPS games. It provides basic functionality.

Unity Standard Assets Standard assets that come with unity and contain often used post processing effects, shaders and scripts

UnityVS <http://unityvs.com/>
Plugin for Unity and Visual Studio that allows debugging of unity applications within Visual Studio.

12.2. Software Usage Instructions

Using Oculus Rift

Install Oculus Rift Runtime 6.0 or higher from Oculus Rift Developer site. Set up the device as detailed in Oculus Rift usage instructions.

You do not need to plug in and set up Oculus Rift camera hardware – the software application does not support it.

To rotate the player character inside the application just rotate your head while wearing Oculus Rift.

For more detailed instructions on how to use Oculus Rift see the official website if Oculus VR.

Using Delta Six

Plug in Delta Six Bluetooth receiver into the PC. Set the receiver to PC mode by pressing the middle button.

Assemble Delta Six. Scope and silencer are optional parts – they are not necessary for correct function of the application.

Turn on Delta Six. Hold it in a neutral position and hold the calibration button until it vibrates signaling it has been calibrated.

If there is a need to recalibrate the Delta Six during the game press and hold the software calibration button on the Delta Six (see the next chapter), reposition your Delta Six and release the button.

Moving the Delta Six inside the game will move the Crosshair image and change the shooting direction. Aim the Delta Six so the crosshair image is on the target you wish to shoot and press the trigger button to shoot.

To move your character inside the game, use the analogue stick on the Delta Six.

To reload your gun, tap the ammo clip on the Delta Six.

To activate the sniper function, lean onto Delta Six as if you were trying to look through the Scope.

Button Actions



Figure 29 - Delta Six button mappings

Jump by pressing **F3** button on the keyboard or **A/X** button on Delta Six.

Crouch by pressing **F4** button on your keyboard or **B/O** button on Delta six.

Reload by pressing **F2** on your keyboard or **X/□** on your Delta Six. You can also **tap the ammo clip** on your Delta Six to reload.

Select options in menus by pressing **F3** button on your keyboard or **A/X** on your Delta Six.

Cancel conversations or close menus by pressing **F1** button on your keyboard or **Y/Δ** on your Delta Six.



Figure 30 – Main gameplay buttons on Delta Six

Fire by pressing **Left Control** on your keyboard or **Trigger/RT** button on Delta Six.

Activate/deactivate sniper mode by pressing **Z** button on the keyboard or lean over your Delta Six as if you were going to **look through the scope**. Sniper mode makes the game zoom in on target making it easier to shoot targets that are far away.

Move your character and navigate through menus by pressing **WASD** buttons on your keyboard or use the **analogue stick** on your Delta Six.



Figure 31 - Analogue stick and Start button on Delta Six

Activate software calibration by holding **F5** or **G** button on your keyboard or **R3** or **RB** on your Delta Six, reposition your Delta Six and release the pressed button.

Pause the game / activate the menu by pressing **Escape** key on your keyboard or **Start** button on your Delta Six.

Using the main application

Recommended hardware specification to run the application is the following:

- Nvidia GTX 970 / AMD 290 equivalent or greater
- Intel i5-4590 equivalent or greater
- 8GB+ RAM
- Compatible HDMI 1.3 video output
- 2x USB 3.0 ports
- Windows 7 SP1 or newer

To run the application produced as a part of this project locate the directory with the application and run `DiplomskiRad.exe`. If you have Oculus Rift active on your system, the application will automatically open on your Oculus Rift, otherwise it will open on your monitor set to the resolution of your monitor and full screened.

The first scene will load shortly allowing the player to choose whether to play tutorial or main level without or with stressors.

Tutorial scene is a level designed to teach new players how to play the game. Returning players should skip it and choose to play the main level instead.

Choosing Main Level without Stressors allows the players to play the main level. In this mode no stressors will activate.

Choosing Main Level with stressors loads the identical level as the previous choice, however, in this mode stressors will activate and interrupt the gameplay.

Tutorial Level

Tutorial level teaches players how to play. Follow the in game instructions to finish the tutorial.

Main Level

To finish the level all enemy targets must be neutralized. The player can check the number of remaining targets at any time by pausing the game.

While the game is paused the player can also choose to exit the game, reset the level or play the tutorial level instead.