A
Project Report
on

QUANTUM RUSH

In partial fulfillment of the requirements
for the award of the degree
Of

MASTER OF COMPUTER SCIENCE

By
Harshvardhan Rajpurohit
200485741

Department of Graduate studies
University of Regina
3737 Wascana Pkwy,
Regina, SK S4S 0A2

April, 2024

Abstract

Quantum Rush VR is inspired by the popular first-person shooting game SuperHot. It aims to offer players a distinctive blend of action-packed gameplay and mindfulness. The VR experience is set in a visually stunning and minimalistic environment. The catch of the gameplay "Time moves only when the player moves" leads to intense combat situations that require strategic thinking. The time control feature grants players the ability to perceive and react to their surroundings with heightened awareness and precision. By slowing down time to a near standstill during moments of stillness, players can carefully plan their actions, dodge incoming threats, and execute strategic maneuvers with unrivaled control. For the development of the final project, various assets such as the XR Interaction Toolkit, Oculus Hands, and Weapons Pack were considered.

Figure 1: A still from SuperHot VR

Contents

Abstract 2
Glossary 4
Introduction 5
Project Methodology and System Overview 6
  Development Approach 6
  System Architecture 6
  Components and Interactions 8
    Player 8
    Enemies 9
    Interactables 10
    Non-Interactables 11
    Scenes 11
    User Interface Elements 12
  Implementation Details 12
    Locomotion System 12
    Enemy AI system 14
    Time Control System 16
User Interface Design 18
Conclusions and Future work 19
References 20

Table of Figures

Figure 1: A still from SuperHot VR 2
Figure 2: A still from Quantum Rush 7
Figure 3: Hands game object used to denote the player. 9
Figure 4: A still of in-game enemy 10
Figure 5: All the interactables accessible in-game. 11
Figure 6: User Interface in the Game 12
Figure 7: XR Origin components. 13
Figure 8: Code snippet for NavMesh Agent 15
Figure 9: Code snippet for time control feature 17
Figure 10: In-game user interface menu. 18
Figure 11: A still from Quantum Rush 19

Glossary

Below are the commonly used terms in the report.

1. *Assets*: Representation of any item or game object that can be used in your application. It can be anything ranging from an image file, a script file to models or animations.
2. *Prefabricated Items (Prefabs)*: These are a special type of component that allows fully configured game objects to be saved in the Project for reuse. These game objects have all the necessary components applied to them and can be shared between scenes or even other projects.
3. *Script*: These are behavior components that can be applied to game objects and modified in the Unity Inspector. These contain the actual C# code that is executed during the gameplay. They can manipulate anything and everything be it the game object they are attached to or the whole application settings of the project.
4. *Time scale*: Unity game engine works on a time scale which is a scale at which time passes in the game engine. Manipulating the value of this component can be used for slow motion effects or to speed up the application.
5. *Interactable*: Every item that the player can interact with or use with in the game is interactable. In other words, every weapon is to be considered as an interactable object that can be utilized in-game. As per the game engine, every game object with an interactable script attached as a component to it will be affected by an interactor (player).
6. *Interactor*: The player or the game objects representing player's features like hands or body that will be controlled by the player to interact with the interactable are termed to be interactors.
7. *XR integration toolkit*: It is a high-level, component-based, interaction system for creating VR and AR experiences. It provides a framework that makes 3D and UI interactions available from Unity input events.
8. *Time Control Mechanics*: The core gameplay feature of the SuperHot VR project, allowing players to manipulate time by moving, slowing it down to nearly a standstill, providing them with heightened awareness and strategic advantages. Technically, the movement of headset and the two controllers are considered, and time is manipulated based on their movements.

Introduction

SuperHot VR is a first-person shooting game that incorporates the use of time control mechanics as seen in movies like The Matrix. The aim of the project is to put the players in a position where they can control the time in the game world, thus giving them enough freedom to play the game at their own pace. The idea is to let the players study the surroundings and plan their moves ahead of time to clear out the horde of enemies surrounding them. I always fancied the idea of controlling time as per my needs. Thus, the application provided me the perfect place to put that thought into play. We often plan for almost everything beforehand, knowingly, or unknowingly, but there are times when we rely solely on our senses or gut feelings. The game puts the players into such situations where they have split seconds to choose an appropriate approach to solve the problem.

This project serves as a test for both my problem-solving and coding abilities, as well as my creative skills. The use of Unity game engine, along with certain specific prefabricated items, are used for the project. For the assets I could not find a working prefabricated item for, I was forced to design them by myself using specific software applications, for instance, Blender for modeling and animation. The idea of time control mechanics does seem easy to implement theoretically, but when applied to practical work, it does seem to have its problems to take care of. The usage of the time scale component of the game engine is how the slow-motion effects are achieved. The animator component was used for handling the animations and the transitions between them. Unity's Navmesh class, along with the inbuilt physics system, was used to design the working of the opponents, specifically their movements and their reaction to the surroundings, respectively.

The following sections of the report explain each of these components' workings in detail. The project methodology and system overview explain the overall workings of the project, ranging from the development model that was used to the final build and system architecture details. The system architecture section explains the workings of the whole system and individual components briefly. The implementation details section explains the execution of the time control, the locomotion or movement of player and enemies, and the enemy AI. The user interface design section contains efficient information about the user interface and the instructions and directions of usage. Lastly, additions to the current system and its features are discussed in the conclusion section, explaining in-depth about the future improvements to the project.

Project Methodology and System Overview

Development Approach

Considering the short size and duration of time for the project, the decision to use an incremental model as the appropriate development approach was administered. The Incremental Model is a software development process that divides requirements into separate modules. In this model, every module progresses through phases such as requirements, design, implementation, and testing. Every successive release of the module either adds a new function or updates the existing ones to the previous released versions. This continues until the desired system state is achieved. Since the game settles in the genre of first-person shooting game, the first and foremost module to be developed for the application was the shooting mechanics. Before that, the mechanics to grab and hold a particular object in a virtual world needed to be implemented to get the system at a state to implement the shooting mechanics. Once the shooting mechanics were implemented, the next main module to work upon was the enemy system, which included the responsiveness to the shooting bullets, their navigation to the player and their spawn in any given level. The successive module was focused upon the time control mechanics that were solely dependant upon the locomotion system of the player. The last module was essentially clubbing everything together to form levels and figuring out the progression system between levels along with the animations, audio effects and transitions wherever necessary.

System Architecture

The project shares multiple similarities to the original game "SuperHot VR", right from the time control mechanism to the way the enemies are depicted on the screen. The use of Unity game engine was incorporated since the original game was built using the same engine, thus leveraging its robust features in the field of VR applications development. Since Unity game engine supports C# programming language thus the use of the same was a viable choice for implementing the core mechanics of the application. The use of Visual Studio was considered for the coding, debugging, and testing of the code responsible for the core game mechanics.

The project was essentially targeted for Oculus devices and thus will be most compatible with the quest or rift devices, nevertheless the basic need of VR device irrespective of any type is a must. Any device with a minimum resolution of 1440 x 1600 pixels per eye, 64 GB memory and 6 to 8 GB of RAM comprises of the minimum requirements for the smooth gameplay. The build has been only set for android devices thus a need of VR headset and controllers is a must to have the best experience while gaming. The use of XR interaction toolkit allows the usage of inbuilt tracking devices in the head mounted display as well as either of the controllers which enables precise tracking of player movements and interactions.

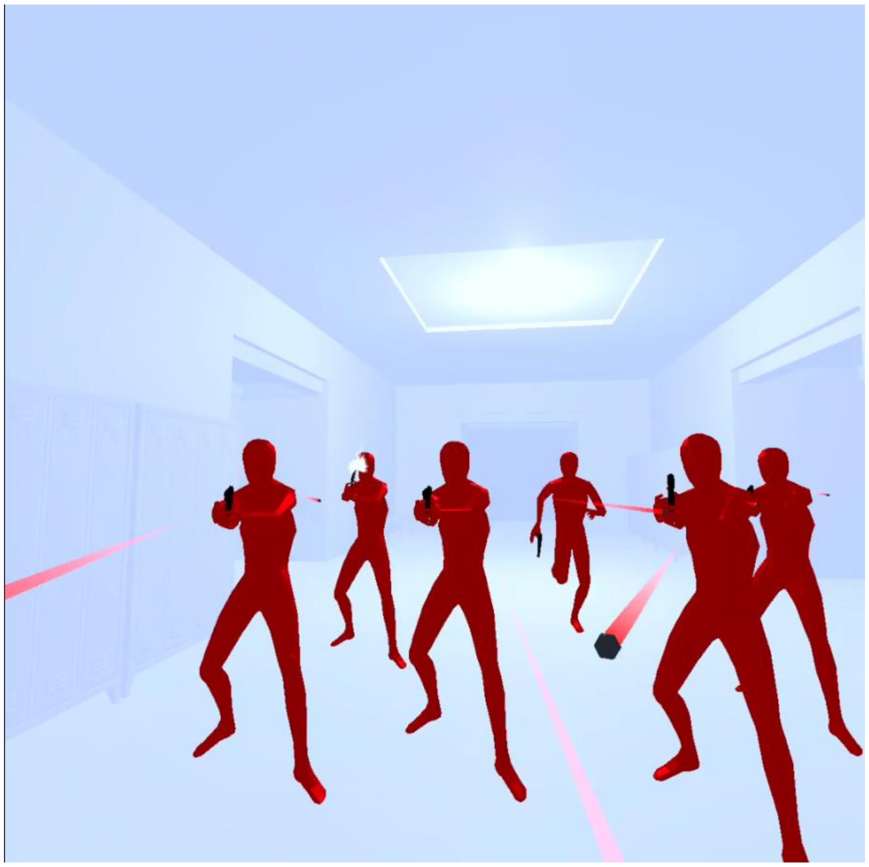


Figure 2: A still from Quantum Rush

The aim of the project is to provide an immersive gameplay experience that blurs the line between reality and virtuality, allowing players to engage in intense, time-manipulating combat scenarios. As mentioned earlier, the main goal is to provide players with a freedom of controlling time and planning their actions beforehand to bring a resolution to difficult situations. While adapting the gameplay mechanics for VR, the project strives to maintain the core identity and essence of original SuperHot series by including its distinctive art style and strategic gameplay. With a variety of weapons available at every level, players have a choice that affects their gameplay drastically. For instance, choosing a baseball bat against an enemy wielding a gun might appear much more challenging than using a gun against the same enemy. Every weapon is designed to be a throwable object thus providing the player a little boost in terms of dodging incoming horde of enemies. Bullets can either be dodged or be blocked using the weapons wielded. If the player fails to do either of them, it is assumed that the bullet hit the player and the level resets, and the player must start all over again. Every level starts with a densely fog environment with a red cube appearing right in front player. Grabbing the cube reveals the level and thus starts the game. There are 2 types of levels designed, namely Classic and Endless. As the name suggests, endless indicates the type of level where the enemies keep on spawning while the player aims to clear them all out without getting shot or hit. The limit for enemy spawns has been set to a 100 but can be altered according to needs of the players, whereas this limit is set to a constant value in the classic type. The classic type basically contains levels that the player needs to clear to complete the game. For the implementation of this project, 3 different levels with their unique orientation were crafted. And after completion of each level, the environment resets the fog and changes to another level.

Components and Interactions

The Oculus Quest headset uses an inside-out tracking system for the tracking mechanism. Every controller contains a set of infrared LEDs located on the rings of the controller. The headset has 4 infrared cameras located at different locations in an efficient manner such that they continuously take images and use them to efficiently map the tracking to virtual objects. Every virtual object that the user notices on the headset screen needs to be generated using the Unity game engine. There are several assets, both prebuilt and self made, that were used for the construction of these virtual objects and essentially the complete gameplay. They are described as follows:

Player

The idea of first-person shooter incorporates that the player views the action through the eyes of the character they are controlling. Thus, the use of headset provides the eyesight for the character to be controlled and the controllers aid in tracking the movements of the hands. No virtual object has been rendered for any body part of the player except their hands. The use of a prebuild asset called Oculus Hands was used to highlight the hands of the player. The use of different components from the XR interaction toolkit are utilised for controlling the steady tracked movements of these virtual hands.

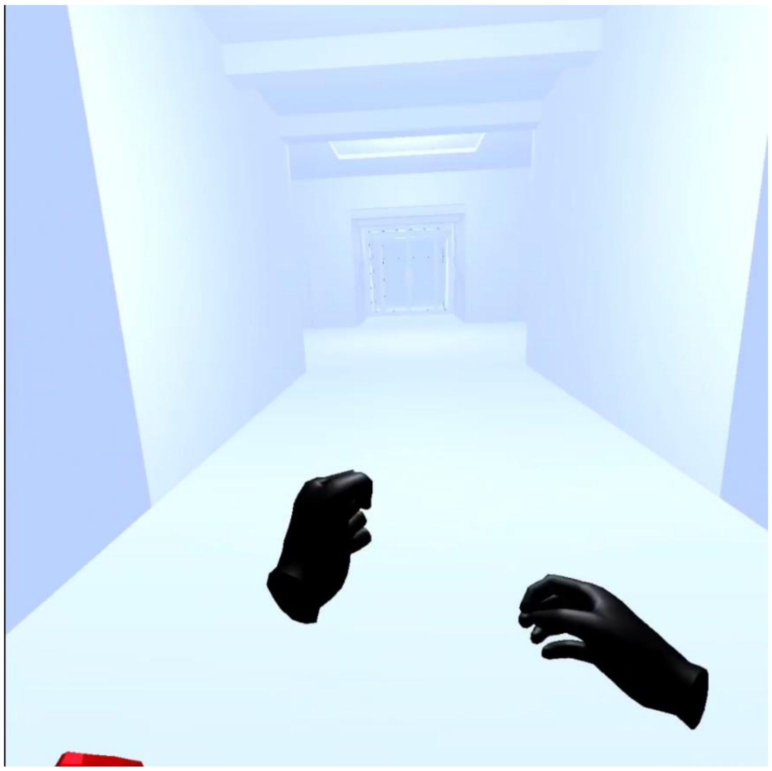


Figure 3: Hands game object used to denote the player.

Enemies

One of the key features of the SuperHot VR game were the enemies with their striking red glass-like opaque humanoid appearance. They are pure red and have a low-poly mesh (a polygon mesh with relatively small number of polygons). There are 2 different types of enemies that are designed in the project namely, *Fighter* and *Shooter*. As the name suggests, the Fighter types do not wield any weapon and will strike the player with their bare hands. On the contrast, the Shooter types have a gun attached to their hands, which they utilise to shoot the player. It takes one punch, one hit or one bullet to kill an enemy and upon death, they leave behind their weapon.

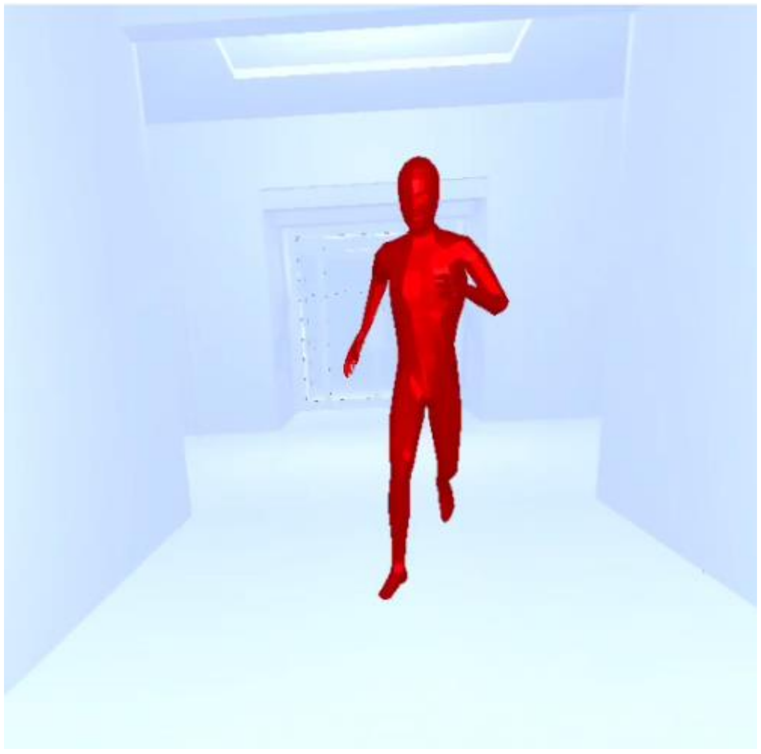


Figure 4: A still of in-game enemy.

Interactables

This highlights all those virtual objects that the user can interact with throughout the gameplay. Starting from the red cube, that initializes the game to the weapons that can be used against the enemies. All the interactables have a similar material like the enemies except they have black color, except the red cube and each of them are a grabbable virtual object. There are 4 different interactables, 3 of which are the weapons, and the last one is the level initializer. A baseball bat, a knife and a gun comprise of the weapons interactable while the red colored interactable cube is used as a level initializer.

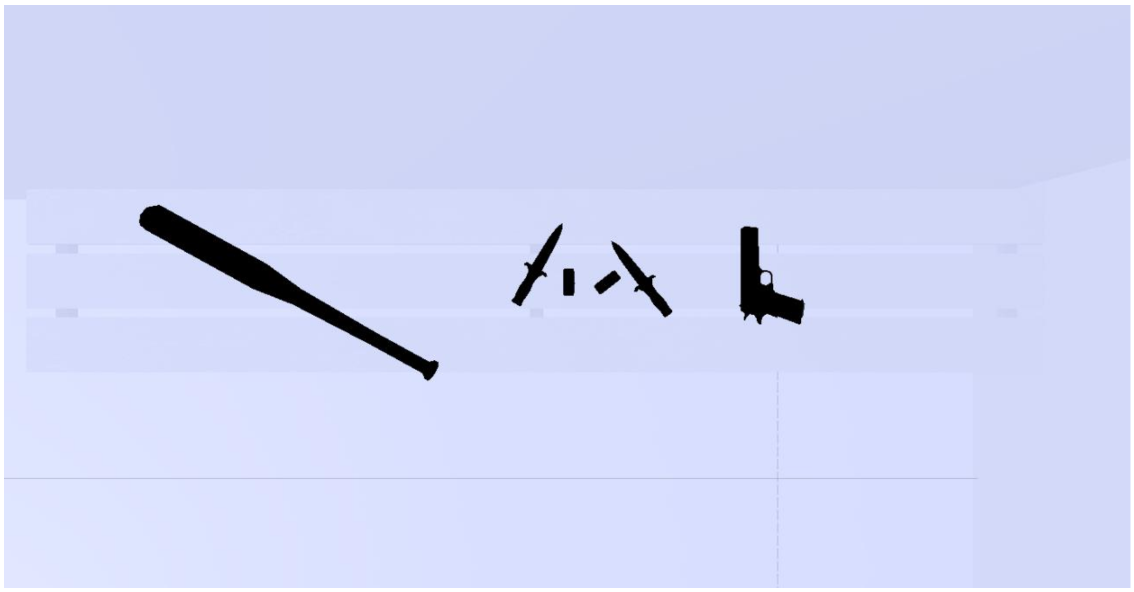


Figure 5: All the interactables accessible in-game.

Non-Interactables

Every other item except the above defined virtual components are termed to be non-interactable virtual objects. These are the objects that essentially form the environment and the obstacles of the level, ranging from walls to chairs and other objects that the user can see and be affected by but cannot directly interact with. Interaction here means the capability to grab and move objects in the virtual world. For the level design, various assets like bench or locker are used as non-interactable objects. The other distinguished feature of non-interactables is their plain white color. This not only complies with the unique visualization method of the game obstacles and environment in the virtual world but reduces the confusion between the accessibility of the objects to the player, hence directing the focus onto required part of the gameplay.

Scenes

Scenes are basically like folders or containers that fundamentally contain all the elements and game objects of the application. Scenes are the way of incorporating the idea of levels in the application. These can also be used for containing UI components to form a menu for the application. They help you organize your application in chunks and promotes the use of modularity and reusability. They also facilitate transitions between parts of the game, such as menu screens or cutscenes, thus allowing the developer to create a seamless and immersive experience. Unity’s scene management supports techniques like streaming or asynchronous loading, enabling dynamic loading and unloading of scenes which can be used for complex environments. This project incorporates the use of 5 such scenes, 4 environment scenes while 1 UI scene to handle the user interaction for initializing the game.

User Interface Elements

SuperHot VR has a unique approach to main menu and other UI elements. The original game uses game objects representing CD or floppy disks to highlight different levels that the player chooses to play by mounting on a drive on the system in front. The project does not have any implementation of interaction of menu through virtual game objects instead uses the basic menu items provided by the unity game engine. The menu has been set as an individual game object thus it does not appear to be an overlay obstructing the vision of the player. The use of ray interactors was incorporated to allow the player to use the controllers to select options from the menu.

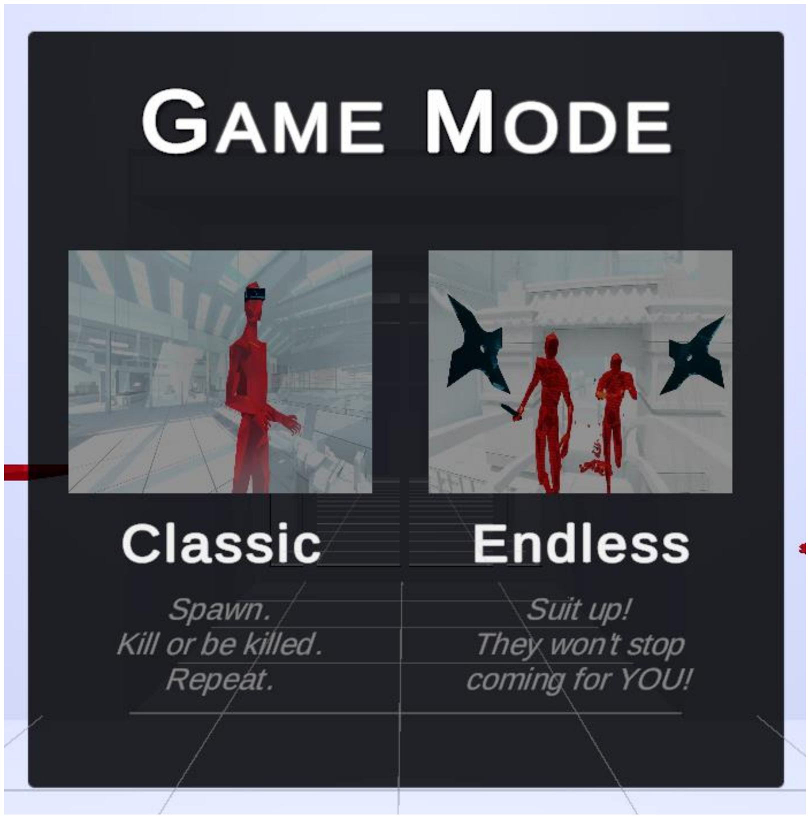
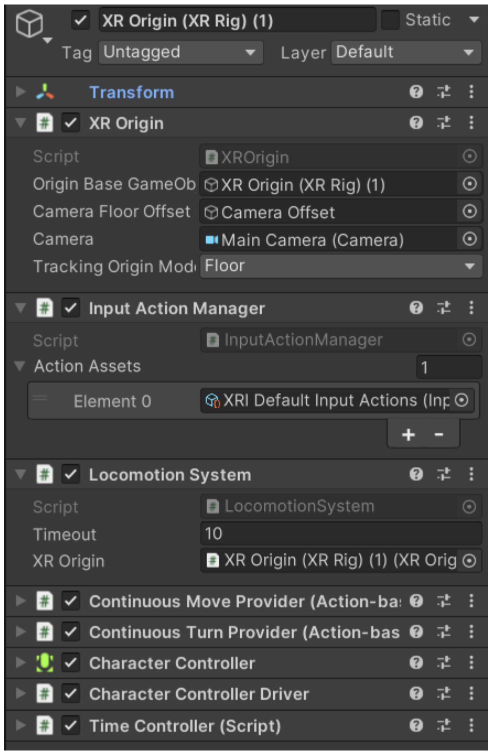

GAME MODE
Classic
Spawn. Kill or be killed. Repeat.
Endless
Suit up! They won't stop coming for YOU!

Figure 6: User Interface in the Game

Implementation Details

Locomotion System

Locomotion system here refers to the movement of player in the project. This section of the report explains the implementation of the mechanics in the project briefly using the XR interaction toolkit. The XR interaction toolkit is a high-level, component-based interaction system that aids in creation VR and AR experiences. It provides a framework that makes 3D and UI interactions available from Unity input events. With the use of infrared cameras and tracked motion of controllers the virtual hands of the player can mimic their movements in the application but the movement of the character around the level could not be incorporated just by the infrared cameras. One of the approaches was to allow the player to move in a physical space and to essentially capture and replicate the movement to the virtual object but the constraint of availability of space seems to affect it badly. The second approach was to use the thumbstick on either of the controller to govern the movement of character. Although this movement would be an artificial and affect the immersion of player but could guarantee the smooth locomotion around the map. The implementation of both the approaches has been made for the project such that the player has a choice however they want to experience the gameplay.



```
XR Origin (XR Rig) (1)                         Static
Tag Untagged                    Layer Default
Transform
XR Origin
  Script                       XROrigin
  Origin Base GameOb           XR Origin (XR Rig) (1)
  Camera Floor Offset          Camera Offset
  Camera                       Main Camera (Camera)
  Tracking Origin Mod          Floor
Input Action Manager
  Script                       InputActionManager
  Action Assets                1
    Element 0                  XRI Default Input Actions (Inp
                                                    + -
Locomotion System
  Script                       LocomotionSystem
  Timeout                      10
  XR Origin                    XR Origin (XR Rig) (1) (XR Orig
Continuous Move Provider (Action-ba
Continuous Turn Provider (Action-bas
Character Controller
Character Controller Driver
Time Controller (Script)
```

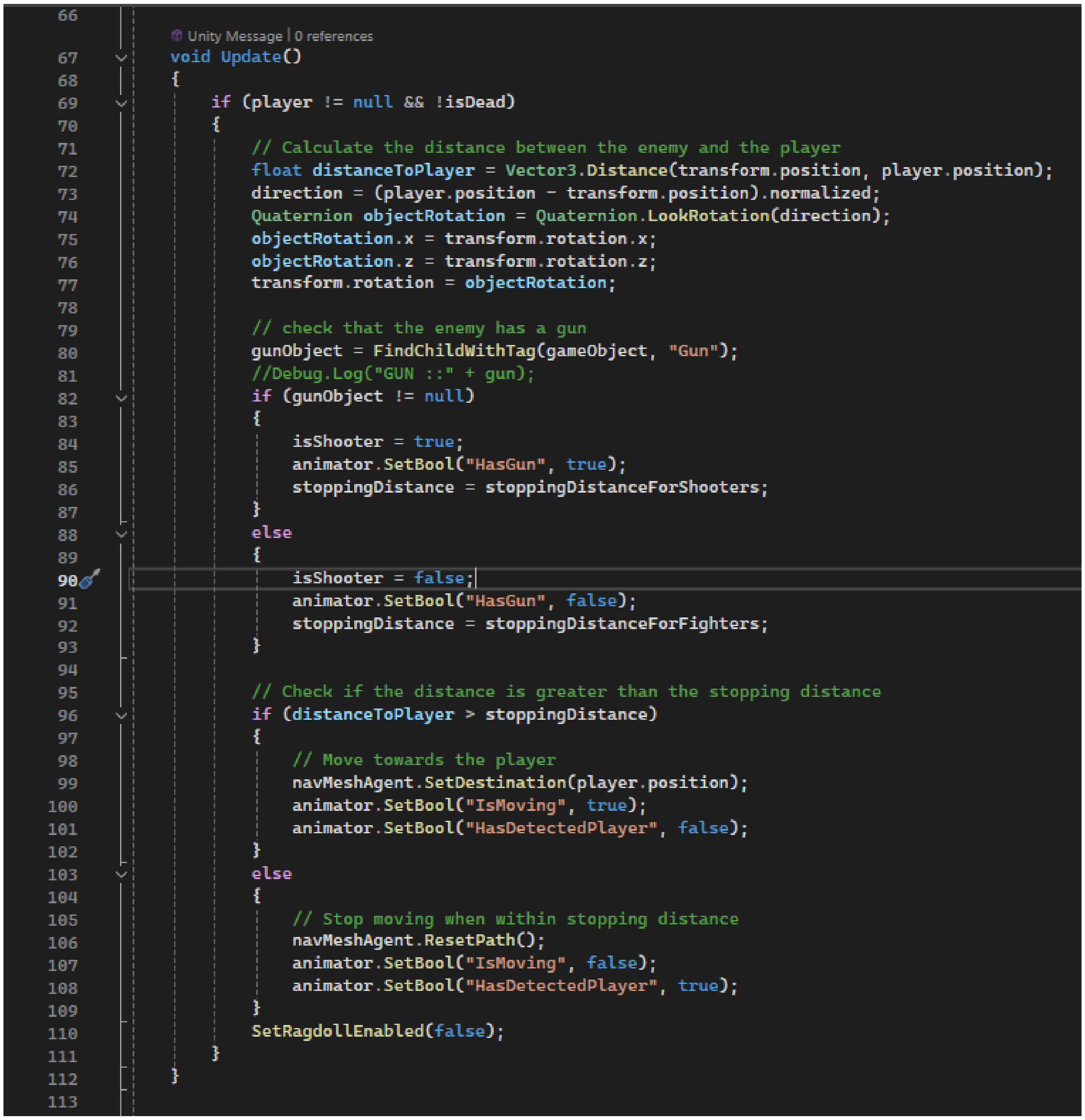
Figure 7: XR Origin components.

The XR interaction toolkit provides the necessary prefabs and components required for handling the tracking and motion of objects. The XR Origin component as shown in the figure above is the main driver for the locomotion of the player. The input action manager is responsible for handling all the inputs from the controllers and mapping them to their desired bindings. The locomotion component is the script essentially handling the movement of game object governed by the inputs from controller. The options in XR origin component handles the tracking of headset and provide the required information to the input action manager to map them with the game objects. The character controller component along with track pose driver allows the efficient tracking of headset be it the elevation from the ground of the rotation or movement. Further use of components like Continuous move or turn provider and the character controller adds more dynamic to the locomotion. The continuous move or turn provider uses the existing locomotion system to efficiently map the action input to move or rotate the game object by the specified values in the component. Both the component allows a smooth movement of the game object. An alternative to the continuous turn provider is the snap-turn provider that allows the rotation of the object by certain degrees every time the corresponding input is triggered. Meaning, if a value of 45 degrees is used for the snap-turn provider component, with every activation of the corresponding input, the game object would turn to a specific direction by 45 degrees.

Enemy AI system

This section explains the implementation of the enemies in the project, be it their movement or their reaction to the different objects in the game environment. The enemy models can be identified by their striking red appearance and their willingness to hit the player via all means possible. The enemy AI system is responsible for their locomotion from their spawn location to the player's location, their interactivity with different game objects in the environment and their reaction when hit with an interactable. For their locomotion, the use of a navigation mesh was implemented using the Navmesh class provided by Unity game engine. Whereas for the reactivity with the environment and other game objects, the use of physics engine was incorporated. A ragdoll system was introduced that was applied to every game object representing an enemy. This ragdoll system was responsible for disabling the interactivity of the enemies once hit by any interactable. Various visual and sound cues were also added to show the reactivity of the enemy with the interactables.

A navigation mesh, or NavMesh, is a designated mesh in any unity scene, which highlights navigable areas in the game environment, including the areas where the characters can walk, as well as obstacles. The enemy object has a Nav Mesh Agent component attached to them, that helps them access the information provided by the Nav Mesh for figuring out a way to reach the player. The *Modified Enemy script* component is a script that handles the usage of these Nav Mesh components. The *Stopping distance for Fighters* and *Stopping distance for shooters* are the governing input factors for specifying the distance the enemy needs to stop from the player. The code snippet highlighted in the figure below shows actual logic for the movement of enemies. In every update cycle, the distance and the rotation to the player is calculated from the position of the game object currently. The use of stopping distance is to check whether the enemy has reached within the proximity of the player to deal damage. With the use of existing methods, such as *SetDestination* and *ResetPath* from the Nav Mesh agent class allows the locomotion for the enemy game object. Since the script is common for both the *fighter* and the *shooter* enemy types, additional check on the weapon object, specifically *gunObject* has been added to the script since that governs the usage of the stopping distance.



```
Unity Message | 0 references
void Update()
{
    if (player != null && !isDead)
    {
        // Calculate the distance between the enemy and the player
        float distanceToPlayer = Vector3.Distance(transform.position, player.position);
        direction = (player.position - transform.position).normalized;
        Quaternion objectRotation = Quaternion.LookRotation(direction);
        objectRotation.x = transform.rotation.x;
        objectRotation.z = transform.rotation.z;
        transform.rotation = objectRotation;

        // check that the enemy has a gun
        gunObject = FindChildWithTag(gameObject, "Gun");
        //Debug.Log("GUN ::" + gun);
        if (gunObject != null)
        {
            isShooter = true;
            animator.SetBool("HasGun", true);
            stoppingDistance = stoppingDistanceForShooters;
        }
        else
        {
            isShooter = false;
            animator.SetBool("HasGun", false);
            stoppingDistance = stoppingDistanceForFighters;
        }

        // Check if the distance is greater than the stopping distance
        if (distanceToPlayer > stoppingDistance)
        {
            // Move towards the player
            navMeshAgent.SetDestination(player.position);
            animator.SetBool("IsMoving", true);
            animator.SetBool("HasDetectedPlayer", false);
        }
        else
        {
            // Stop moving when within stopping distance
            navMeshAgent.ResetPath();
            animator.SetBool("IsMoving", false);
            animator.SetBool("HasDetectedPlayer", true);
        }
        SetRagdollEnabled(false);
    }
}
```

Figure 8: Code snippet for NavMesh Agent

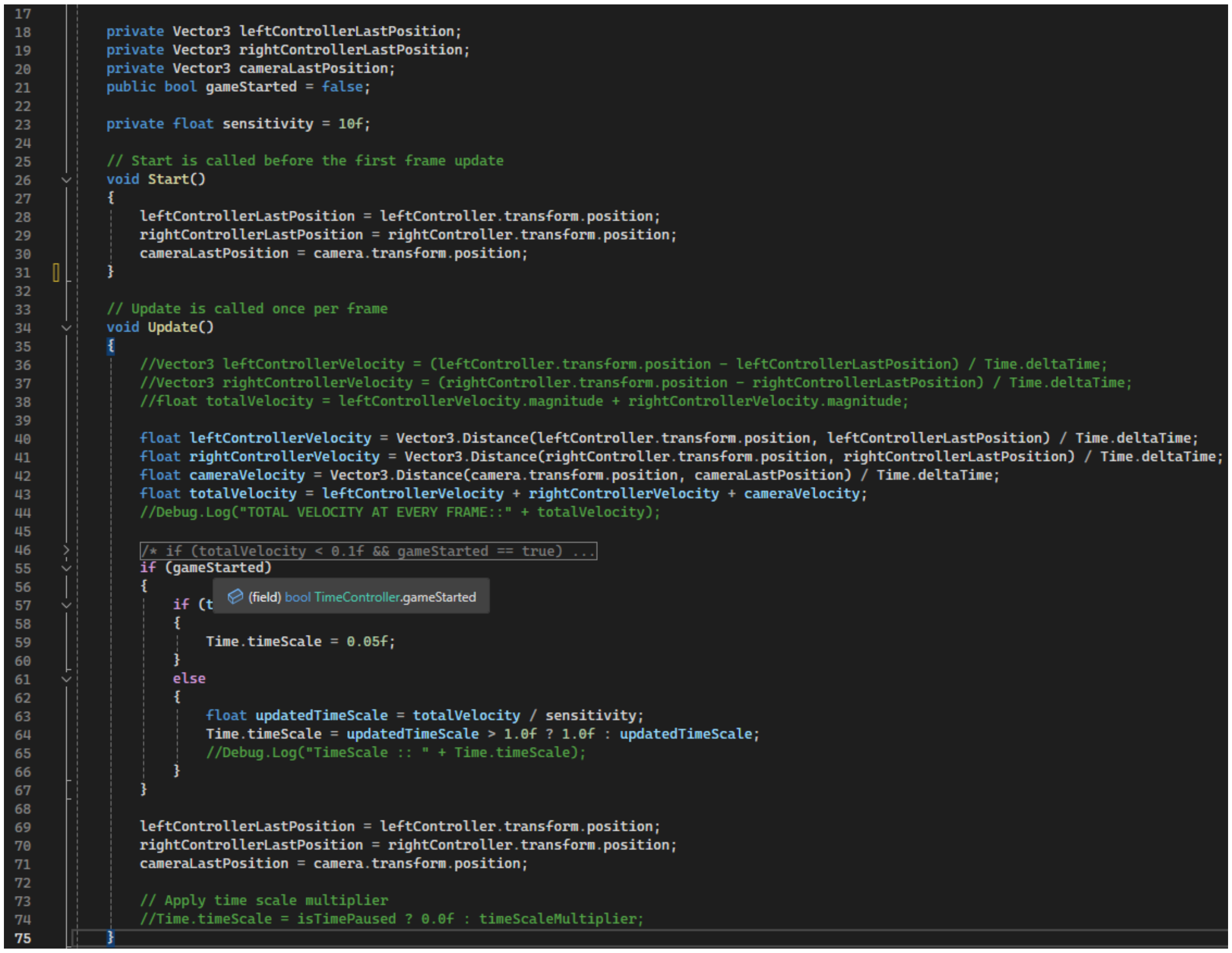
Ragdolls are the variants of animated virtual objects whose bones are completely taken over by the force of physics. The ragdoll system is responsible for efficiently disabling the enemy locomotion and all the working of the game object in general to mimic the action of being dead. The use of collision events along with the physics subsystem is used to effectively activate this ragdoll system. The ragdoll system needs an active rig, which means that individual game objects, highlighting different limbs of the character, needs to be mapped to their individual properties in the ragdoll system wizard. Simply by dragging the game objects to their respective properties, the ragdoll system generates the colliders, rigid bodies and joints that make up the ragdoll. Initially all the game objects as set as kinematic objects, they are not affected by the ragdoll system, as soon as a collision with an interactable occurs, the objects are no longer set to be kinematic and thus act under the influence of ragdoll. Whenever a hit is recorded, after the ragdoll system is activated, a small amount of impulse force is added to the individual game object that recorded the hit to mimic the impact of the hit on the game object and allow ragdoll to move the object effectively.

The enemies are spawned at specific location located by a spawner object. A spawn script was developed to spawn the number of enemies with a delay between each spawn. The use of coroutine is incorporated into spawner script to perform the operation over time instead of instantly. Each level has its own spawners which spawn enemies after the level has been initiated. These spawners also have a level manager script attached that efficiently keeps track of the total enemies in the level, along with the alive and dead count. Once the dead count equals the total count, that's when the script triggers the fog system to activate and a cutscene to efficiently let the player transition between levels. A script associated with every individual limb object of the enemies is responsible for detecting the collision from an interactable object. This script is responsible for both ragdoll system activation as well as registering a dead enemy onto the counter. The check of dead count with total count is executed at every update cycle thus no lag in the execution of functionality is observed. The combination of all the above briefed sub-systems effectively creates an enemy which is capable of tracking and hunting down the player.

Time Control System

What sets the game apart from rest of the first-person shooting games is the time control mechanics that lets the player control the pace of the gameplay. The time control feature provides players the ability to react to their surroundings with heightened awareness and precision. Experienced players can clear out the levels within seconds while the novice players take their time figuring and planning everything out before the execution phase. The *Time Manager* class provided by the Unity game engine lets the developers control and manipulate time related properties and actions. Specifically, the *timeScale* component provided by the *Time Manager* class that decides the scale at which time passes in the application. The value of 1.0 indicates time passes at the same rate as real time, a higher value would increase this factor thus depicting a fast-forward effect in the application while a lower value would make the application appear to be in slow motion. A value of 0.5 would indicate that the time passes 2x slower than real time. Since it is a global property, it applies to every game object in the scene equally thus not requiring handling anything explicitly. The mechanics in the original game is based on the motion of the controllers and the headset. The faster their motion is the more the game tends to run on a normal time scale and vice versa. At every update cycle, the positions of either of the controllers along with the headset are fetched and a displacement is calculated. The change in the displacements indicates the motion and thus increase the timescale value accordingly. This would essentially mean that if no displacement or motion was detected, the time scale value would be nearing to 0 thus making the application appear in a paused state.

With the collection of every briefed system in harmony, the project is effectively capable of incorporating a gameplay like the SuperHot VR application. Although the project can use up more smooth systems and interactions to enhance the experience, however it captures the core of the original game play efficiently with the time control mechanics. This project provides the perfect base to enhance the gameplay further thus improving the experience.



```
    private Vector3 leftControllerLastPosition;
    private Vector3 rightControllerLastPosition;
    private Vector3 cameraLastPosition;
    public bool gameStarted = false;

    private float sensitivity = 10f;

    // Start is called before the first frame update
    void Start()
    {
        leftControllerLastPosition = leftController.transform.position;
        rightControllerLastPosition = rightController.transform.position;
        cameraLastPosition = camera.transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        //Vector3 leftControllerVelocity = (leftController.transform.position - leftControllerLastPosition) / Time.deltaTime;
        //Vector3 rightControllerVelocity = (rightController.transform.position - rightControllerLastPosition) / Time.deltaTime;
        //float totalVelocity = leftControllerVelocity.magnitude + rightControllerVelocity.magnitude;

        float leftControllerVelocity = Vector3.Distance(leftController.transform.position, leftControllerLastPosition) / Time.deltaTime;
        float rightControllerVelocity = Vector3.Distance(rightController.transform.position, rightControllerLastPosition) / Time.deltaTime;
        float cameraVelocity = Vector3.Distance(camera.transform.position, cameraLastPosition) / Time.deltaTime;
        float totalVelocity = leftControllerVelocity + rightControllerVelocity + cameraVelocity;
        //Debug.Log("TOTAL VELOCITY AT EVERY FRAME::" + totalVelocity);

        /* if (totalVelocity < 0.1f && gameStarted == true) ...
        if (gameStarted)
        {
            if (t  (field) bool TimeController.gameStarted
            {
                Time.timeScale = 0.05f;
            }
            else
            {
                float updatedTimeScale = totalVelocity / sensitivity;
                Time.timeScale = updatedTimeScale > 1.0f ? 1.0f : updatedTimeScale;
                //Debug.Log("TimeScale :: " + Time.timeScale);
            }
        }

        leftControllerLastPosition = leftController.transform.position;
        rightControllerLastPosition = rightController.transform.position;
        cameraLastPosition = camera.transform.position;

        // Apply time scale multiplier
        //Time.timeScale = isTimePaused ? 0.0f : timeScaleMultiplier;
    }
```

Figure 9: Code snippet for time control feature.

User Interface Design

User Interface plays an essential role in navigating the player between scenes or levels. Therefore, it needs to be developed efficiently to not effect the gameplay and ruin the experience of the player. The project uses only 1 UI menu that the user can use to choose between the type of gameplay he wants to experience. The use of Unity UI toolkit is taken into consideration for developing the user interfaces. The UI uses the basic components such as the button and text elements on a panel to display the necessary information. The use of a ray interactor is required to allow the player to use this UI without the need to interact with the elements by touching them.

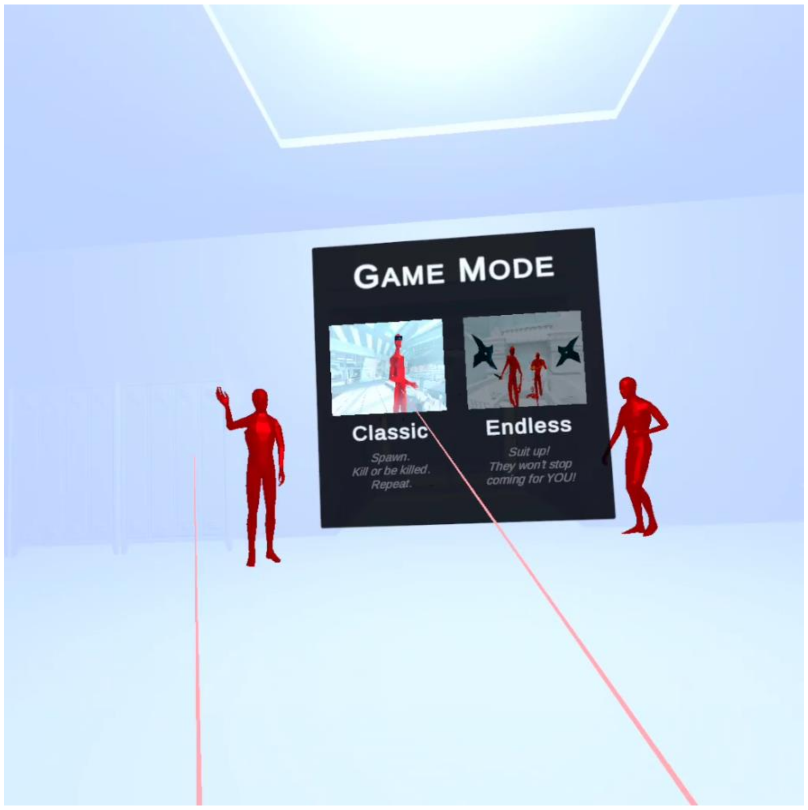

GAME MODE
Classic
Spawn. Kill or be killed. Repeat.
Endless
Suit up! They won't stop coming for YOU!

Figure 10: In-game user interface menu.

The project can be launched like any other application in the system the player uses. The necessary step before using the system is making sure the system is properly set and calibrated for the best experience. The controls for the system are simple and uses the select button on either of the controllers to establish a selection of object in the system. The only additional step that differs from the original game is the usage of an interactable. The project basically simulates the idea of holding an object in the virtual world by incorporating the use of buttons situated on the handle of the controllers. For holding a virtual interactable object, the player must keep this button held and use the select button to activate the working of the object (only applicable if it is a gun). Necessary precautions and steps must be taken to stay aware of the physical surroundings to avoid accidents. To best enjoy the experience regular breaks should be taken to avoid discomfort and fatigue.

Conclusions and Future work

In conclusion, this project offers an immersive and engaging virtual reality experience mindful of the popular title Superhot VR. Through the careful integration of natural controls, captivating gameplay mechanics, and visually stunning environments, players are transported into a dynamic world where time manipulation and strategic thinking are of prime importance. Furthermore, there are several routes for further enhancement. Drastic refinement of gameplay mechanics, including the implementation of new features and challenges, can enrich the player experience. Additionally, efforts to optimize performance across different VR platforms will be crucial for maximizing impact. For the future work, there are several exciting opportunities for future development of the project to include new levels and challenges, feasibility of adding multiplayer features and new methods of engaging with the items in the game environment to make the experience much more pleasant. Drastic changes need to be made to the overall gameplay of the system to incorporate more diversity thus aiming to increase the immersion of the player.

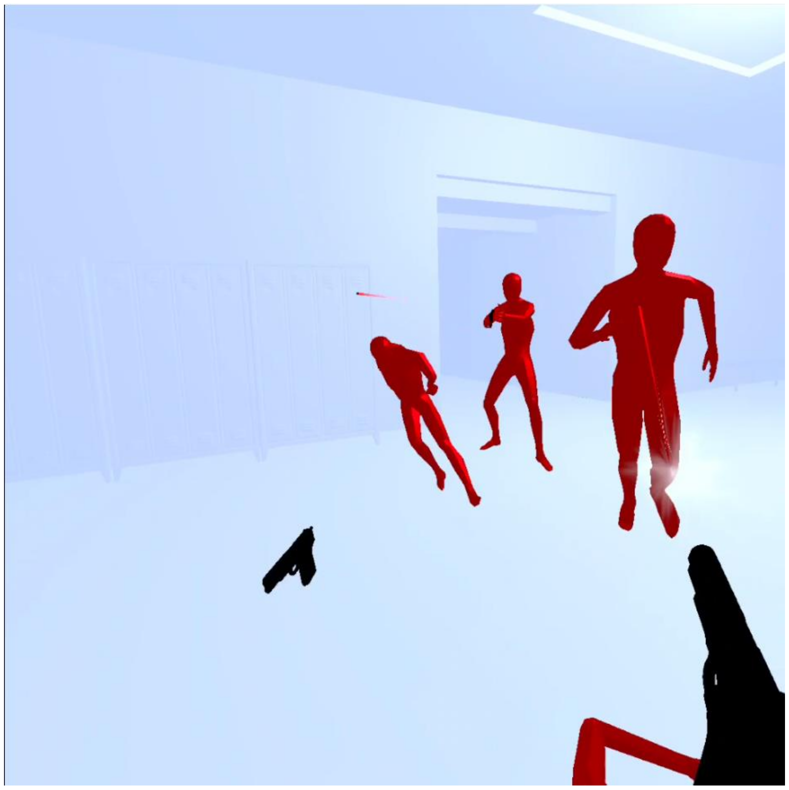


Figure 11: A still from Quantum Rush

References

- Oculus. (2023). Oculus Integration [3D toolkit]. Retrieved from https://assetstore.unity.com/packages/tools/integration/oculus-integration-deprecated-82022
- Ithappy. (2023). Weapons FREE [3D Model]. Retrieved from https://assetstore.unity.com/packages/3d/props/weapons/weapons-free-260492
- Nokobot. (2020). Modern Guns: Handgun [3D Model]. Retrieved from https://assetstore.unity.com/packages/3d/props/guns/modern-guns-handgun-129821
- VIS Games. (2022). Locker Room Props [3D Model]. Retrieved from https://assetstore.unity.com/packages/3d/props/interior/locker-room-props-3355
- Zijin Li. (2022). "To VR or to PC? : Comparing Player Experience and Physiological Reaction in Superhot VR vs Superhot PC". Game Science and Design Thesis at Northeastern University. Retrieved from https://www.proquest.com/docview/2728512719/fulltextPDF/F98FBB8DE3924EA6PQ/1?accountid=13480&sourcetype=Dissertations%20&%20Theses