

**Senior Design Project**

**The Design of a Virtual Reality Game for STEM Education**

ENGE476 Senior Design Project I

Department of Engineering and Aviation Sciences

University of Maryland, Eastern Shore

Tyler Gantz

Project Advisor, Dr. Lei Zhang

Submitted

December 7th, 2018

List of Contents

[List of Figures 3](#_Toc453190988)

[Abstract 4](#_Toc453190989)

[1. Introduction 5](#_Toc453190990)

[1.1 Backgound/Motivation 6](#_Toc453190991)

[1.2 Objective 6](#_Toc453190992)

[1.3 Design Requirements 6](#_Toc453190993)

[1.4 Design Constraints 6](#_Toc453190994)

[1.5 Design Methods 6](#_Toc453190995)

2[. Implementation Plan 7](#_Toc453191000)

[2.2 Timeline/Milestones/Delivery Plan 7](#_Toc453191005)

[3. Implementation 8](#_Toc453191006)

[3.1 Implementation of Task 1. 8](#_Toc453191007)

[3.2 Implementation of Task 2. 9](#_Toc453191008)

[3.3 Implementation of Task3. 11](#_Toc453191008)

[4. Conclusion. 15](#_Toc453191009)

[Acknowledgement 16](#_Toc453191010)

[Appendix 17](#_Toc453191011)

[REFERENCES 19](#_Toc453191018)

List of Figures

[Figure 1. Game-Based Learning Logo 5](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501622)

[Figure 2. First Call of Duty vs. Newest Call of Duty: Growth of Technology in Visuals 5](#_Toc7501623)

[Figure 3. Versatility in Implementation of Virtual Reality 6](#_Toc7501624)

[Figure 4. Illustration of the different elements game--based learning incorporates 6](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501625)

[Figure 5. Flow diagram displaying the design approach 7](#_Toc7501626)

[Figure 6. Project Timeline 9](#_Toc7501627)

[Figure 7. Unity Logo 10](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501628)

[Figure 8. Unreal Engine Logo 10](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501629)

[Figure 9. Scene used during engine configuration 10](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501630)

[Figure 10. Picture supporting the proposed alien spaceship setting 11](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501631)

[Figure 11. Gameplay flow diagram 11](#_Toc7501632)

[Figure 12. Examples of some assets found in the game 12](#_Toc7501633)

[Figure 13. SteamVR Objects 13](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501634)

[Figure 14. Code for the body collider object script 13](#_Toc7501635)

[Figure 15. Picture of the in-game hand models 14](#_Toc7501636)

[Figure 16. Picture in each lens vs. Perceived image 14](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501637)

[Figure 17. Picture of a single floor piece 15](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501638)

[Figure 18. Overview of the entire room 16](#_Toc7501639)

[Figure 19. LCD Screen object 16](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501640)

[Figure 20. Cover screen, question screen, and correct screen. 17](#_Toc7501641)

[Figure 21. View of Multiple-Choice Question Buttons, and full-screen object view 18](#_Toc7501642)

[Figure 22. Picture of some of the keys found in the game 19](#_Toc7501643)

[Figure 23. Hologram found next to Question 1 19](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501644)

[Figure 24. Multiple Choice Key Slot and Associated Code 20](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501645)

[Figure 25. Battery used for circuit design objectives 20](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501646)

[Figure 26. A 5700 Ohm Resistor 21](#_Toc7501647)

[Figure 27. Battery port 21](file:///F:\Senior%20Design%20Report%2004.16.2019.docx#_Toc7501648)

[Figure 28. Example of the wiring and resistor ports 22](#_Toc7501649)

[Figure 29. Energy shield examples 23](#_Toc7501650)

Abstract

By the end of the project, summarize the project into short text and put here.

1. Introduction

## Background/Motivation

From the beginning of any engineering program, students are required to take difficult courses that involve new math and physics concepts. From experience, this can be overwhelming, as a lot of these concepts are foreign both mathematically and mentally. Visualizing topics such as circuit configurations or truss problems in statics are not always intuitive to new students.

As it stands, the standard curriculum for engineering students at UMES consists of just lectures with a handful of labs included in a four-year degree program. While this proves to be successful for most students, more learning tools are always welcome. Now, what if those learning tools broke away from the monotony of the typical lecture, where students sit quietly and take notes, with the only interaction being asking questions to a professor. What if they provided an alternative to a lab, which typically only has students flipping a switch, measuring a value, then flip another switch, measure that value. In other words, what if there was a learning tool available to students that is both fun and interactive, but still educational.

This is where game-based learning comes in. Game-based learning, or GBL, uses competitive exercises, either pitting students against each other or getting them to challenge themselves in order to motivate them to learn better. Incorporating GBL into a curriculum has shown to motivate students to learn and, in turn, learn more effectively. Games provide an element of fantasy to the players, something that is not present in the typical lecture or lab. This element is what provides a fun and memorable experience for the students playing, which is what makes this approach to education so effective. [1]

Figure 1. Game-Based Learning Logo

Technology in gaming has come a long way since its beginning back in the ’80s. Visually, games have evolved from the 8-bit, pixelated graphics to the stunning 4K graphics found in games today. The way the players control and play their games has also evolved from using simple game 8 button gamepads to full motion controls. However, there is one innovation that stands out among the rest, and that’s virtual reality (VR). Especially in recent years, VR has become a prominent part of gaming with the release of popular hardware, like the HTC Vive and the Oculus Rift. Gaming platforms, such as PlayStation, Steam, and various mobile applications all offer VR support with compatible hardware. Having this technology at our disposal is great, but is it really that vital to incorporate it rather than just developing a simple video game?  

Figure 2. First Call of Duty vs. Newest Call of Duty: Growth of Technology in Visuals

In the Merriam-Webster Dictionary, virtual reality is defined as an artificial environment which is experienced through sensory stimuli (such as sights and sounds) provided by a computer and in which one’s actions partially determine what happens in the environment. From this definition, one can tell that VR offers something that no 3-D game on a computer screen can offer, and that is true immersion. It is because of these immersive capabilities that virtual reality has proven to be an effective method in teaching and providing experience for users in situations that would not normally be easily accessible or reproduced. For example, VR is being used in sports to improve certain aspects of players’ games, such as situational awareness or correction in body mechanics. A specific example comes from a study that had shown that football players were able to shave off a second of their decision making in accordance to a defensive coverage by simulating these situations and testing them on how to handle it. Another example is of Kelly Oubre Jr., a player for the Washington Wizards in the NBA. "I really saw a difference in my jump shot and free throws," says 20-year-old wing Kelly Oubre, who grew up playing "Call of Duty" and is used to wearing a headset. "I could see my mechanics, what I needed to do right." Oubre's true shooting percentage is up this year, from 50.7 to 53.4.[2] This technology is literally changing people’s careers for the better, so applying it to education for engineers is most definitely worth it.

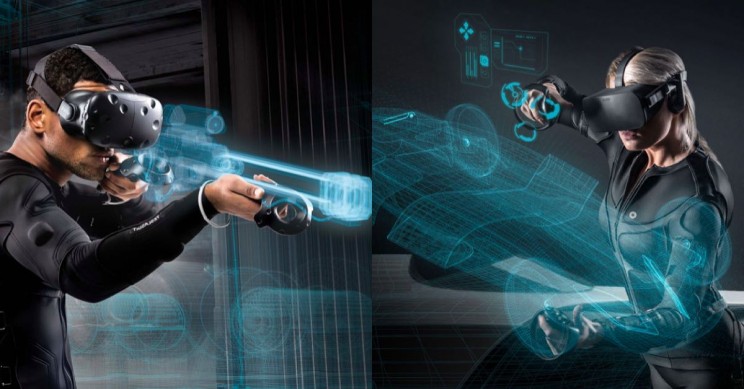


Figure 3. Versatility in Implementation of Virtual Reality

It is obvious that a combination of education, gaming, and virtual reality can provide a powerful learning tool for students if done correctly. While school subjects being the main part of a game does not sound like the most enticing idea, using the concepts of game-based learning correctly and productively could prove that to be false. Combining different gaming elements like action, shooting, puzzles, and the creative implementation of visual and sound effects with educational topics can create an experience where the player is having as much fun as a noneducational game while learning at the same time. Games provide a sense of competition that could drive students to try harder to learn subjects so they can succeed in the game, in turn, leading to success in the classroom. This is a learning tool that needs to be taken seriously and given a chance in schools or at home today.

Figure 4. Illustration of the different elements game--based learning incorporates

## Objective

The objective of this project is to develop an effective game-based, virtual reality learning tool for electrical engineering students.

## Design Requirements

1. Educate and entertain through the immersive features of VR gaming.
2. Completable within 20 minutes
3. Cover 3 subjects in the Electrical Engineering field: DC Circuits, AC Circuits, and Digital Circuits.

## Design Constraints

1. The game will be developed for the HTC Vive.

2. The game must be playable within the minimum PC hardware and space requirements provided by HTC:

* Computer:
* GPU: Nvidia GeForce GTX 970, AMD Radeon R9 290 equivalent or better  
  CPU: Intel i5-4590, AMD FX 8350 equivalent or better  
  RAM: 4 GB or more  
  Video Output: HDMI 1.4, DisplayPort 1.2 or newer  
  USB Port: 1x USB 2.0 or better port  
  Operating System: Windows 7 SP1, Windows 8.1 or later, Windows 10

**Play area:** 2m x 1.5m (6ft. 6in. x 5 ft.

## Design Method

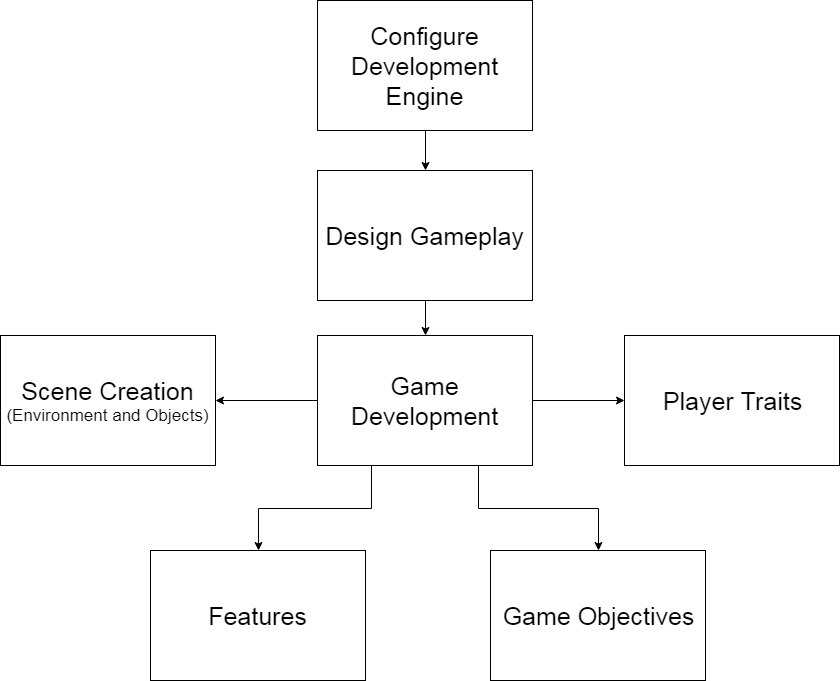


Figure 5. Flow diagram displaying the design approach

1. Implementation Plan

\*This was a solo project designed by Tyler Gantz

## Tasks

* Task 1. Game Development Engine Configuration
  + Subtask 1. Unity vs. Unreal Engine
  + Subtask 2. Configure for development
* Task 2. Design Gameplay
  + Subtask 1. Scripting the game
  + Subtask 2. Create, or source required components to build the game
* Task 3. Development: Scene Development
  + Subtask 1. SteamVR Player Prefab
  + Subtask 2. In-Game Environment
  + Subtask 3. Multiple Choice Question Screens
  + Subtask 4. Keys
  + Subtask 5. Circuit Components
  + Subtask 6. Energy Shields
* Task 4. Development: Player Traits
  + Subtask 1. Health
  + Subtask 2. Damage Output/Gun System
  + Subtask 3. Movement
* Task 5. Development: Features
  + Subtask 1. Heads-Up Display (HUD)
  + Subtask 2. Draw Mode
  + Subtask 3. Multiple Choice Question System
  + Subtask 4. Gun System
  + Subtask 5. Enemies
  + Subtask 6. Audio and Visual effects
* Task 6. Development: Game Objectives
  + Subtask 1. Puzzle 1
  + Subtask 2. Enemy encounter 1
  + Subtask 3. Circuit Design 1
  + Subtask 4. Circuit Design 2/Enemy Encounter 2

## Timeline/Milestones/Delivery Plan

|  |  |  |
| --- | --- | --- |
| **Time** | **Task** | **Comments** |
| **Week 4**  **To**  **Week 8** | Task 1 | Required weeks of deciding on a game engine and configuring Unity after choosing. Provided a smooth transition into developing the game. |
| **Week 4**  **To**  **Week 8** | Task 2 | Designed the gameplay, including setting and objectives |
| **Week 8 to Week 15** | Task 3 | Began development on the game. Designed the majority of scene objects before any programming was done. |
| **Week 16 to**  **Week 17** | Task 4 | Decided important traits for the player, and implemented them into the game. |
| **Week 17 to Week 22** | Task 5 and Task 6 | Finished up development by implementing core gameplay features and objectives. |

Figure 6. Project Timeline

1. Implementation

## Implementation of Task 1. Game Development Engine Configuration

**Subtask 1. Unity vs. Unreal Engine**

The first task of the project required choosing a game engine to work with. The two options at hand were Unity, and Unreal. Some research was done to find what obstacles and details would be presented with either one. Both engines possess a learning curve, but navigating through that curve is different for each engine. From a programming standpoint, Unity games are programmed using C# scripts, while Unreal uses C++. The key difference is that Unity has been available to the public since 2011, while Unreal has only been free and easily attainable for anyone outside of a professional setting since 2015. It is because of this fact that Unity help and information is readily available in books or online from a quick Google search.

Figure 7. Unity Logo

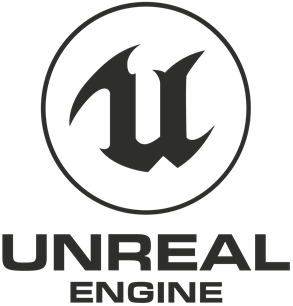
For Unreal Engine, it was widely considered the more powerful engine. This is the engine that some of the largest development companies use because it can achieve higher heights than Unity can visually. One would think that it was an obvious choice to go with the more powerful engine, however, one must also consider the documentation available for Unreal. The level at which the game runs from a functionality standpoint is more important than any level of visual quality. Also, it is well documented that Unreal Engine-powered games require more computing power, which directly conflicts with the minimum PC requirements constraint placed on the project. Considering all the information gathered, Unity was chosen to develop the game.

Figure 8. Unreal Engine Logo

**Subtask 2. Configure Unity for Development**

Once Unity was chosen, it needed to be configured for development. In order to configure the engine, tutorials were completed on scene building. The tutorials provided a workshop for object creation and script writing. Once the tutorials were completed, a pre-built scene was found online, which could be used to test the SteamVR Plugin. The SteamVR plugin is an asset pack found on the Unity Asset Store that provides a multitude of scripts and prefabs that are necessary for the basic functions of the HTC Vive. Scripts for object interaction, like picking up and throwing, and the teleportation feature are examples of the most useful assets in the pack. Applying these scripts to the prebuilt scene allowed for coding-focused experimentation more than taking the time to have to design a functioning scene. This is where an understanding of how the HTC Vive detects inputs through code was formed, which was vital for developing the game.

Figure 9. Scene used during engine configuration

## Implementation of Task 2. Design Gameplay

**Subtask 1. Scripting the Game**

Task 2 was all about designing the game from a creativity standpoint. Throughout the early configuration stages of the project, settings and suitable objectives within a certain setting were brainstormed. After a few weeks of weighing out options, the setting was chosen to be an alien space in outer space. The overarching objective was to escape the alien spaceship. The player is going to be put to the test through multiple choice questions (with an electrical engineering focus), physical objectives, and puzzles. The game is supposed to function like an escape room, which is an activity that has become popular over the recent years where groups enter a room full of puzzles, brainteasers, and scavenger hunts. The group has to solve these within a time limit to successfully complete the challenge. This is the concept that the game will take on. It provides a fun, interactive environment that requires the player to think through each step.

Figure 10. Picture supporting the proposed alien spaceship setting

The diagram below provides a detailed layout of how the game will flow.

A close up of text on a white background

Description automatically generated

Figure 11. Gameplay flow diagram

**Subtask 2. Create or Source the Required Components to Build the Game**

In order to create this setting with its objectives, in-game objects needed to either be made or sourced. Unity allows for objects of basic 3-D shapes to be created and placed throughout the game in any kind of orientation desired. From there, you can add components, such as colliders, rigid bodies, scripts, etc. to the object, providing the user with a lot of customization. For this project, the structural design of each of the rooms, all circuit components, and over 90% of the scripts will be self-made using Unity’s built object creation or 3rd party software.

However, in order to make the highest quality game objects, skills in fields like animating and 3-D modeling are necessary. Unity presents a way for programmers to get by without possessing high-level skills in these disciplines by providing the Unity Asset Store (UAS). The UAS consists of thousands of objects, materials, sounds, animations, etc. that have been made by other users who may have more experience working with these assets. Unity developers can then purchase these assets to then use in their own games. It breaks down a wall that would undoubtedly hold back programmers from developing great games. Here is a list of some of the assets purchased for this project, with a short description of how they are implemented:

* Vast Outer Space Skybox – The skybox, or shell, to the environment that provides the visual sensation of being in outer space
* Sci-fi Styled Modular Pack – A collection of assets themed around a futuristic setting. The majority of the objects used for building the setting were found in this pack, like the battery objects, the floor pieces, windows and doors.
* Force Field Effects – A collection of objects that take on a force field effect that will be used to block off objects from the player.
* Futuristic Weapon Pack – A set of highly detailed weapons to be used in the game as the primary weapons for players in objectives that involve shooting.
* CRT LCD Shader – An asset that provides a screen in which you can attach a video player. Mostly provides a level detail to the game because at a close glance, the player can see lines of pixels as if he/she were looking at a real LCD screen.



Figure 12. Examples of some assets found in the game

## Implementation of Task 3. Development: Scene Development

**Subtask 1. SteamVR Player Prefab**

The first object that needs to be mentioned in the scene is the SteamVR player prefab. This prefab includes object, object components, and scripts every developer needs to get started on a VR game. The screenshot on the right displays the most important objects found in the prefab.

The body collider object contains the components that essentially act as the mid-section of the player in the game. While there is no visible body object, there is a capsule collider component attached to this object that is oriented as if there was a body on the player. This is a necessary component to ensure the player interacts with the other objects in the scene correctly. For example, if this component did not exist, an attacking enemy would have no way of contacting the player. This object also contains a small but important script that establishes the relationship between the height of the capsule collider and the “head” of the player. It is what makes the perception of the player from eyesight to the ground feel realistic in-game.

Figure 13. SteamVR Objects

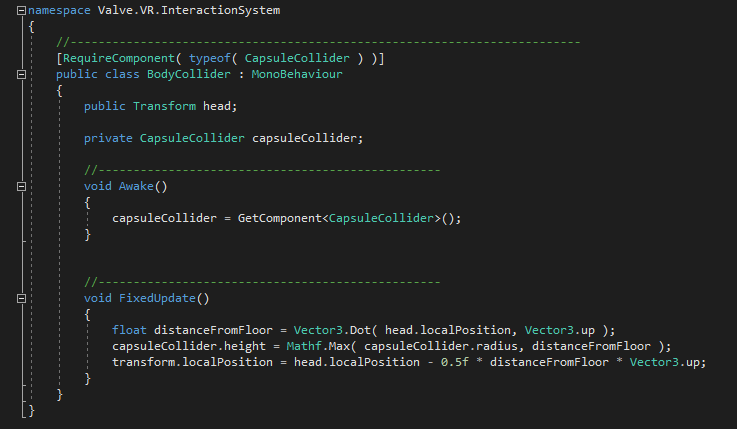


Figure 14. Code for the body collider object script

Next up on the list are the hands of the player. The left and right hands each contain colliders just like the body object, along with some of the most vital scripts for object interaction in-game. By default, the colliders attached to the hands are just spherical colliders placed near the visible hands the player can see in-game. However, 2 modifications were made to the colliders. The first was that the sphere colliders needed to be move and resized so they fit the hand models in-game better. It caused problems when working with interactable objects because the player had to reach too far into the objects to interact with them. The second modification was adding a small capsule collider to more realistically fit the hand models. As seen in the accompanying picture of the hand models, the pointer finger sticks out while the rest of the fingers create a fist. For some of the interactions where precisely contacting objects was unavoidable, it was essential that a collider was added to the pointer finger to add a precision collider to avoid unwanted collisions with the larger-sized sphere collider. Specific examples of scenarios like this will be pointed out later in the report.

A picture containing person, clothing, man, wall

Description automatically generated

Figure 15. Picture of the in-game hand models

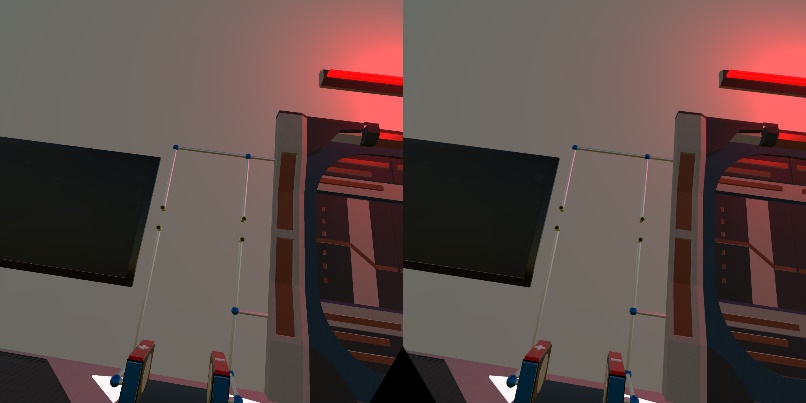
The VRCamera object is what provides the view from the HTC Vive headset. The trait that sets it apart from a typical 3-D game camera is that it has controlled values for stereo separation and stereo convergence. Stereo separation in relation to VR is the distance between each eye. Each eye perceives the world slightly differently due to them being separated from each other, however, the brain perceives the view as one single image. This setting is what creates the immersive feeling in VR, where it feels like your in a new world, rather than looking at a screen inside a headset. It automatically sets the field of view so that the perception is similar to the real world. This perception of seeing one single picture rather than two separate pictures is the result of stereoconvergence. The values for each of these are fine-tuned to create the most realistic view possible for the player and to help avoid eye-strain and motion sickness.

Figure 16. Picture in each lens vs. Perceived image

The [SteamVR] object contains simple but useful scripts. The first script initializes SteamVR on start-up. That ensures SteamVR will open so the player does not have to worry about initializing it themselves and helps avoid communication errors with the equipment. The other script initializes action sets created for the controllers in the game. This is what provides the game the specific input instructions mapped out by the SteamVR plugin for each of the controllers. It is vital in ensuring the game runs correctly with the hardware

**Subtask 2. In-Game Environment**

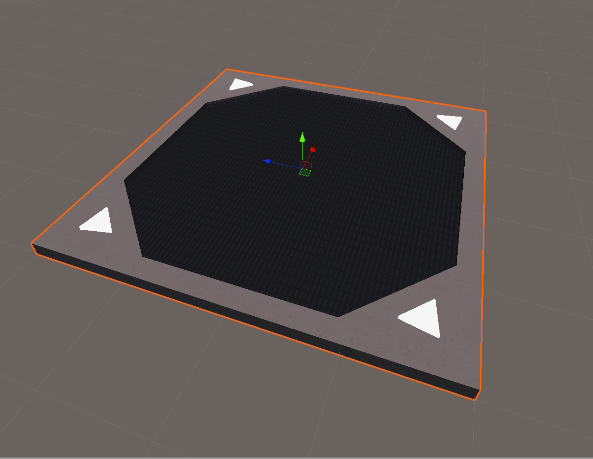
After the idea to create an action escape room game came to mind, it was time to create and build the room itself. A few key points that needed to be adhered to was a lot of space was needed so all features and objectives could fit, it needed to take on a futuristic setting, and it needed to include structures that could be used in fights against enemies. The Sci-Fi Styled Modular Pack (SFSMP) was the go-to asset pack for building the room as it seemed to something to fulfill every need. This was made possible with detailed floor pieces with built-in lights and a lattice cover that scaled to larger sizes nicely. The 1.5-times scaled objects were placed in a 5x5 grid. The room ended up being 225 sq. units in size, which is perceived to be quite large when the player enters the VR world. It was a great start to creating the room.

Figure 17. Picture of a single floor piece

Next, surrounding vertical objects were placed, like walls, doors, and windows. The walls are made up of several thin, but tall and wide cubes made directly in the Unity editor. Doors were placed on the front and far end of the room, one designated to be the door the player was theoretically entered from, while the door on the far end is the one the player needs to open to escape. In each of the 4 corners and in the center of the left and right walls, windows were placed so the player can see into outer space. The corner windows were made out of 3 corner glass panels found in the SFSMP stack on top of each other with some overlap for a decorative effect. The windows in the side walls were created by rotating the big windows found in the same asset pack 90 degrees in the z-direction and then elongating them to match the height of the walls. Once again, each of the objects scaled nicely, which created good visuals to the primitive version of the room.

As previously mentioned, structures needed to be placed in the room that could provide cover during the action sequences in the game. The solution to this was placing 4 pillars, with one in each of the 4 quadrants of the room. The columns are aesthetically pleasing and also block sightlines, which can provide the player safety in enemy fights.

The room obviously needed some decorative elements as well. One of the more impressive looking objects in the SFSMP was the projector. The base object of it looks great, but the projection particle system added a futuristic flair to it. A giant projector was placed in the center of the ceiling, which will act as the system in which the character receives items from allies. The lights included in the asset pack were also great for decoration. There are multiple styles included in the pack. Lights were applied to the left and right side of each pillar, along with on top of the doors.

An overview of the complete room is shown below.



Figure 18. Overview of the entire room

**Subtask 3. Multiple Choice Question Screens**

A repeating objective the player will face in the game are multiple choice questions. In order to create a functional multiple choice question machine, 2 problems needed to be solved: finding a way to successfully display the question and answer choices, and how to allow the player to input an answer.

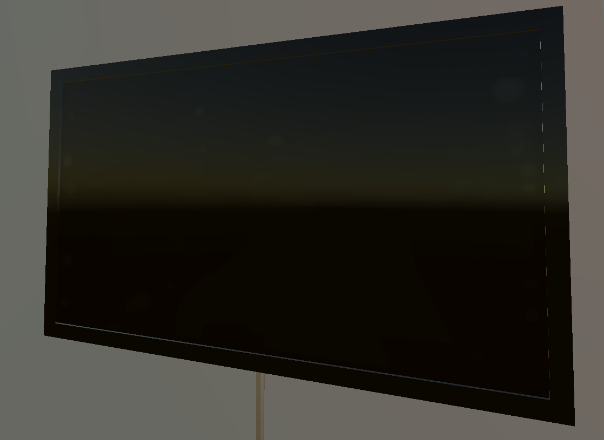
There are multiple ways of displaying text and images in Unity. However, choosing a clean, aesthetically pleasing way to present the questions was the priority. This is where the LCD Widescreen asset came into play. This asset provides an object with an attached shader that is essentially an in-game LCD flat screen TV. Images or videos can be attached to any object as a material in Unity. However, what sets this object apart is the aforementioned shader. The shader on the object creates visible “pixels”, just like one would see if he were to look up close to a real LCD screen. To make each screen look more like a TV, some 3-D cubes were created and elongated and set with a glossy material to act as the trim around the screen.

Figure 19. LCD Screen object

The next step was to create and place all of the on-screen objects found on the multiple choice screens. Unity offers multiple user-interface (UI) objects, a few of which were perfect for this task. The first step was to attach an object called a canvas to the LCD screen object. A canvas is essentially an invisible 2-D base to any UI-based object. The canvas was resized so it fit the shape of the LCD screen. Now that the canvas was placed, the multiple choice images and text could be set up. Each question has 3 screens to transition through, the cover screen that shows the question number, the main question screen, and the correct/incorrect screen. An empty parent object was created for each screen. Then, a UI image object was placed on each and resized to fit the canvas. Each UI image was recolored to black. After each screen was established, the text objects could be placed. One of the key goals of creating these screens was to allow scripts to control what exactly is displayed on each screen. So, at the time of creation, the text objects just included some generic text describing what it will be used for. The cover screen just displays the question number. The main question screen displays the question text, the question timer, and the answer choices. The answer choices use UI button objects for the visual effects, but the script will just change the text inside the button. The correct/incorrect screen just displays whether the player got the question right or wrong. The programming of this feature will be explained later in the report.

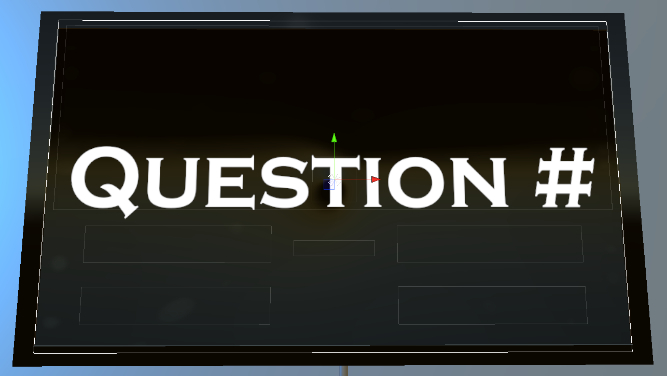
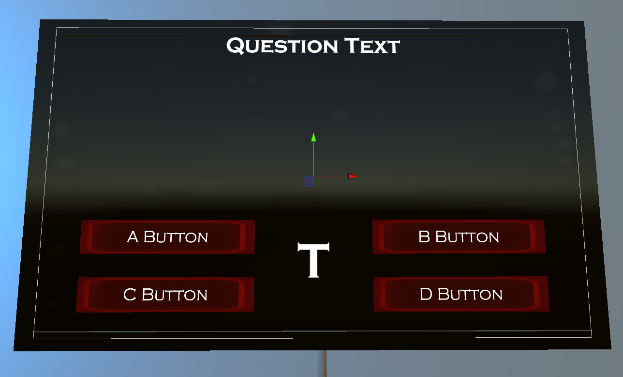
 



Figure 20. Cover screen, question screen, and correct screen.

Now that all of the display objects were created, a solution for taking input from the players was needed. The original idea for this was making the question screen a touch screen. However, the screen would’ve had to be much smaller for the player to get close enough to touch it, while also being able to read it without having to move back from it. We found an asset pack in the store called “Simple Animated 3-D Buttons” that provided just what we needed. Just like the title of the pack states, it provided buttons in 2 different shapes and sizes. However, at first, the buttons only recognized mouse clicks since they were designated for non-VR, 3-D games. After looking into the objects built-in code, all that was needed was changing the *OnClick()* function to the *OnTriggerEnter()* function. Now, when the player comes in contact with the buttons collider, it will run everything inside the function, rather than having to click on it. After getting the buttons to function correctly, they were placed on a table (provided in the SFSMP). One large square button was placed that will be used for starting the question, and then 4 smaller, round

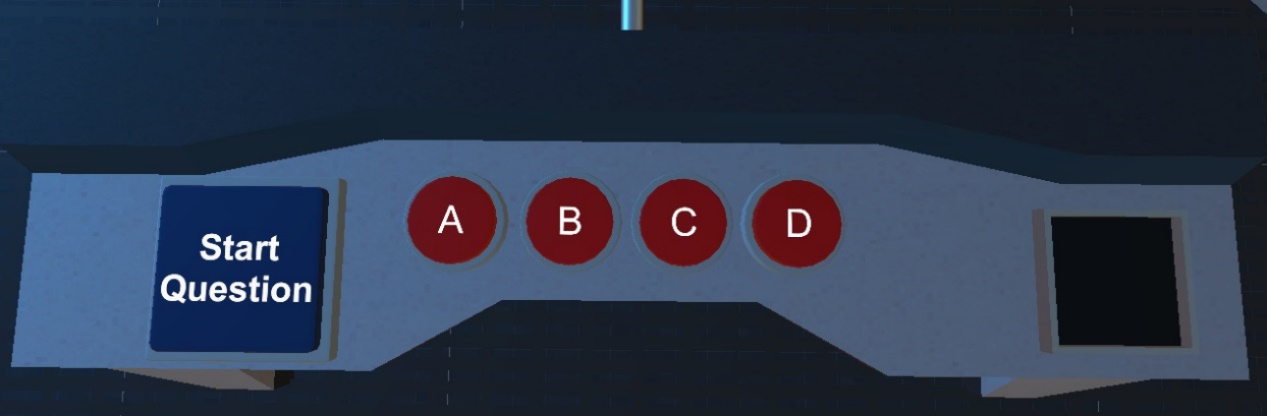
buttons were placed, one for each answer choice. Like stated earlier, the programming portion of this feature will be described later.



Figure 21. View of Multiple-Choice Question Buttons, and full-screen object view

**Subtask 4. Keys**



Figure 22. Picture of some of the keys found in the game

In order to create a better sense of progression in the game, we decided to establish a set of items players need to retrieve from completing objectives. After going through some of the asset packs already in the game, we found these artifacts in the SFSMP, and they were a great solution for the issue. A few objects had to be built into the scene in order to correctly implement this key system. The main function of the keys is to activate a designated multiple choice question. So, we had to create a way for players to know which key goes with which question, and a slot in which the players can place the key to activate the question.

There were multiple ways that we could have gone with displaying the correct keys for each question. For example, the original idea was just to place a picture of the correct key on the screen of each multiple choice question. However, since the game has a futuristic setting, we decided to create a more technologically advanced object to achieve this. Next, to each multiple choice question screen, the player will find what appears to be a hologram version of a key. The key found inside that hologram is the key designated for that question. The hologram key has no collider attached to it, so it is not player-interactable. However, it does have a rigid body and script attached to it so some visual effects could be added to it. The script creates a constant rotational velocity and vertical sinusoidal movement pattern on the hologram, providing a “floaty” visual.

Figure 23. Hologram found next to Question 1

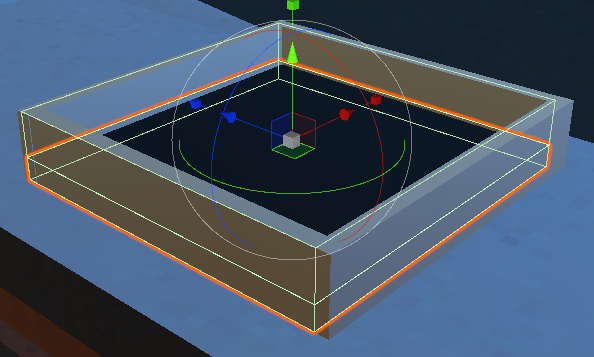
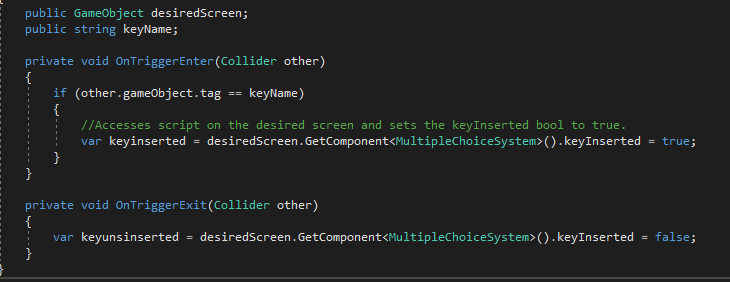
Next, we had to add a key slot to the multiple choice screen set up. These were built using 3-D cubes in the Unity editor. 4 rectangular blocks were oriented into a square shape large enough so that the base of each key will fit. Then, a cube was placed at the base of the slot to provide a flat surface for the key to sit on. This base cube was the most important piece because it is the object the script will be attached to detect if the correct key is inserted. Before the script was written, an extra box collider was attached to the base cube. This collider was moved above the base and resized to about the shape of the inside of the key slot. It was then set as a trigger, so it will not actually collide with anything, however, it will track if something is inside of it. Now that the trigger was set, a script could be written, checking for an object with the correct tag to enter the trigger collider. If all conditions are met, a bool value in the multiple choice script is set to true. That script will be discussed in the multiple choice feature section later. Screenshots of the key slot with highlighted colliders and the code for the key slot are shown below.

Figure 24. Multiple Choice Key Slot and Associated Code

**Subtask 5. Circuit Components**

One of the most important topics in electrical engineering education is circuit design. Understanding how to correctly build circuits under certain conditions is essentially the practical application of everything learned in school. It is because of this that we knew we had to put circuit design objectives in the game. In order to create these objects, we obviously needed to build the circuit components. Since this game was designed mostly as reinforcement to early electrical engineering education, the objectives were mostly centered around basic DC circuits. Therefore, the objects needed were batteries, resistors, and wiring.

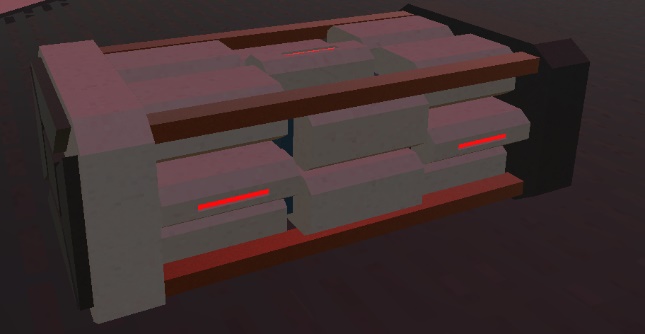
Thanks to the SFSMP, finding a good battery model was rather simple. The pack contains 3 different batteries: small, medium, and large. However, the default size of all of the models was quite large. We ended up choosing the medium battery, but scaled it down to x = 0.1, and y = z = 0.15. Now, we had a solid battery model that made sense in terms of size and fit the scene stylistically.

Figure 25. Battery used for circuit design objectives

Next, we had to make resistors. The resistor models were made completely from scratch using objects and materials in the Unity editor. The resistors can be broken down into 3 parts, the main resistor capsule, the wires, and the colored bands (all of which were placed in a parent object). The main piece of the resistor was created using a capsule object in the Unity 3-D editor. It was resized to x = z = 0.12, and y = 0.2, leaving to be much larger than a realistic resistor, but it would have been impractical and hard to work with if it were that small. Next, the wires on each end were created using a cylinder object, and resized to x = z = 0.03, and y = 0.06. Since they had to protrude down on both ends of the resistor capsule, they were moved to position x = 0.15 and -0.15, and both at y = 0.93. The final part of the resistors were the colored bands. The resistors are 4 band resistors, so the bands represent the first digit, the second digit, the multiplier, and the tolerance of the resistor. They were created by altering 3-D sphere scale values into numbers that would essentially create flat disk-shaped objects. Those scale values were x = 0.02, y = z = 0.125. Then, each band was placed in realistic positions on the resistor capsule. All of which were placed at y = 1, but then spread out along the x-axis at -0.109, -0.078, - 0.046, and 0.096. Now that all pieces of the resistor were placed, materials could be assigned to each. A glossy, tan colored material was placed on the capsule to mock a real resistor's color. A silver metallic material was attached to the wires on each end. For the bands, a glowing, emissive material was made for each possible band color. These materials could be placed as needed on each band depending on the desired resistance.

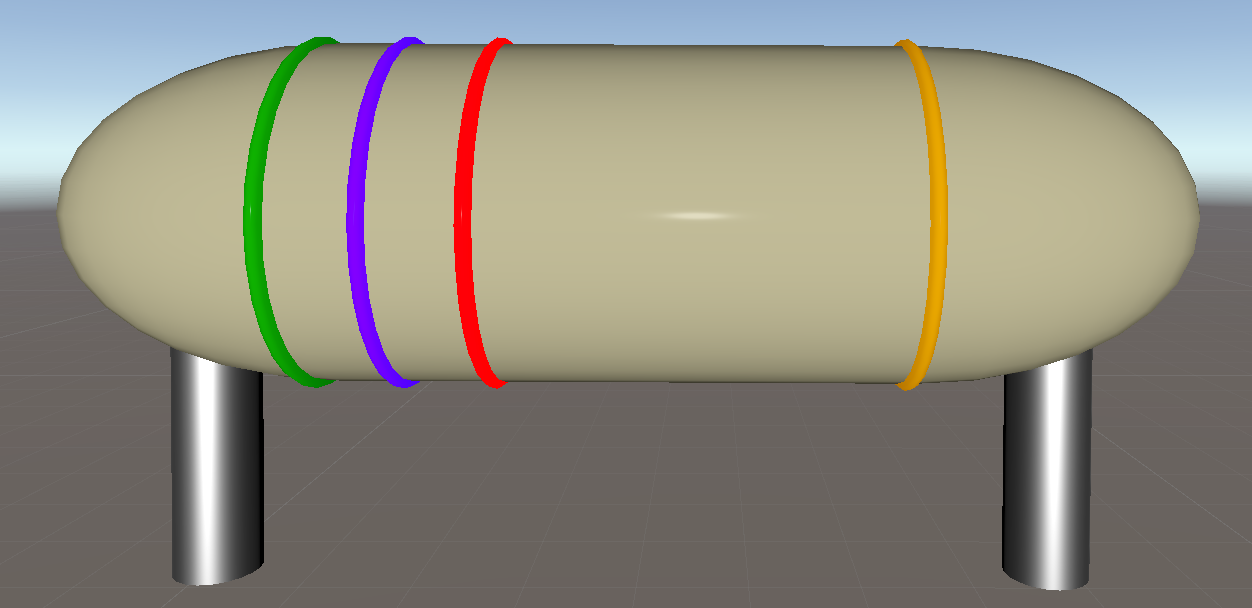


Figure 26. A 5700 Ohm Resistor

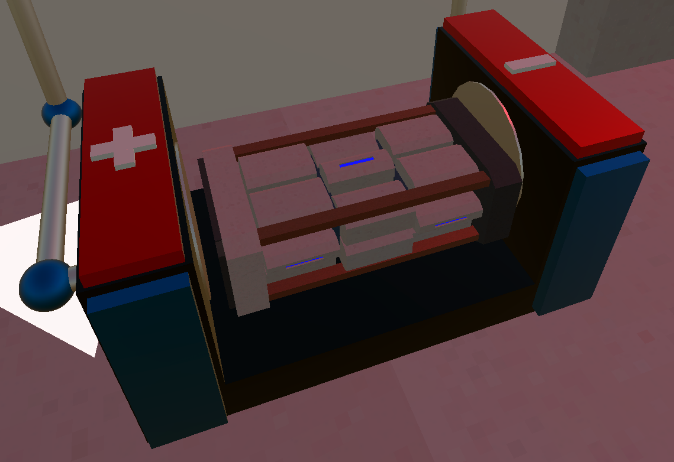
Now that the components were created, the circuits themselves needed to be built. The circuit objects can be broken down into 3 main parts: the battery port, resistor ports, and the wiring. The battery port consists of 3 rectangular shaped cube objects, oriented in a U-shape with the base wide enough to hold a battery. Then, on both the left and right sides of the battery port, blue cube objects protrude out the front, while red cubes protrude out the top to give the object a visually pleasing design. Just like any battery port, the positive and negative terminals needed to be denoted. Once again, using cube-shaped objects, a positive and negative sign were created and attached to the top of each side of the battery port. Wide, but short metallic colored cylinders were placed on the inside of each side of the port to appear as the conductive materials of the port.

Figure 27. Battery port

The resistor ports and wiring are very similar in terms of their base objects. The resistor ports were created using 2 concentric cylinder objects. The outer cylinder objects had a metallic gold material attached, while the inner cylinder had a matte black material, providing the visual of an open cylinder. Unlike typical 3-D modeling software, protruding cuts in objects cannot be done in Unity, which is why this method was chosen. Two of these objects were created at the right distance from each other so the resistor wires can sit in the ports correctly. The wiring was also created using cylinder objects. However, these cylinders have a silver material attached to them, and each piece varied in length depending on the need. A description of the orientation of the wires will be described for each specific circuit when they appear in the report.



Figure 28. Example of the wiring and resistor ports

**Subtask 6. Energy Shields**

Throughout the game, the player will face certain challenges where he/she has to complete an objective to gain access to certain objects. This was an idea for game objectives since the beginning of development, however, deciding how to implement the idea creatively was the difficult part. Since the game takes place in an alien ship, a futuristic, high-tech solution would be the best fit. From past experiences playing futuristic games, a common way to block areas or objects from players was by using some sort of energy shield or barrier. It at the very least blocked access, however, in many instances, it also damaged players. This is exactly what was needed for this game. After searching on the asset store, the “Force Field Effect” pack was found. This asset pack provided force field objects of many different shapes and sizes that could be used to deny access to the players unless they want to take damage. Each force field has a pulsating effect that was created using a particle system. A particle system is a component that can be attached to any object to create a flow of particles, along with a multitude of options to fine-tune the visual effects of it. These energy shields utilized the particle system very well, which made them perfect for the game.

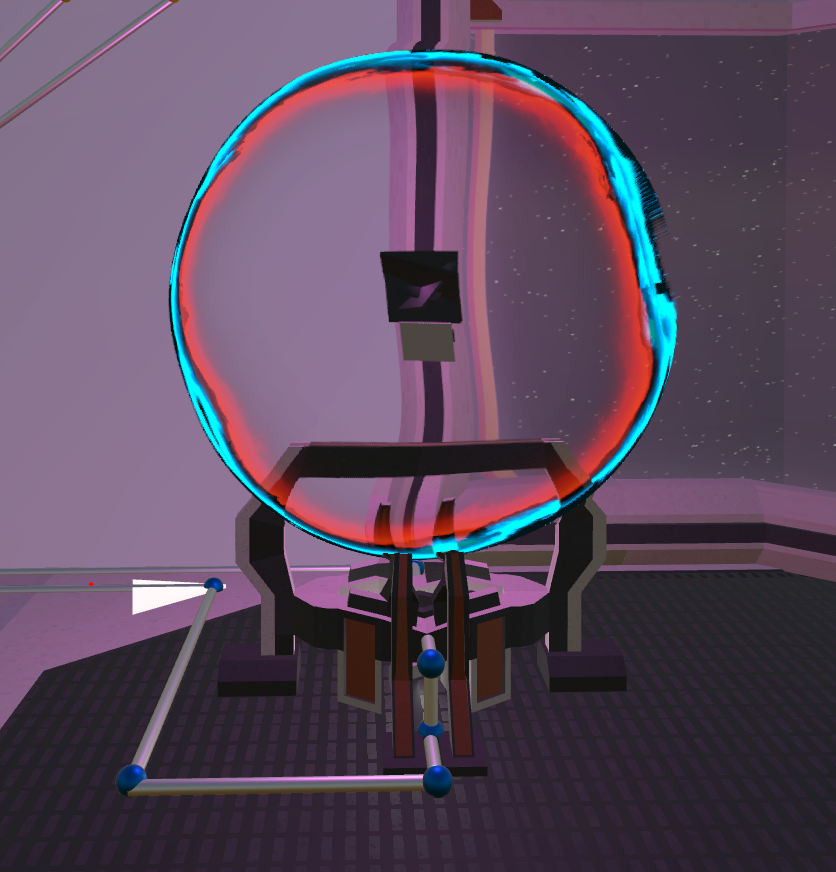
 

Figure 29. Energy shield examples

## Implementation of Task 4. Player Traits

In order to create a good balance between difficulty and fun, certain abilities and limitations need to be placed on the player. These traits were decided early on in development so decisions could be made around what the player can do. The 3 traits used to characterize the player were health, damage output, and movement.

**Subtask 1. Player Health**

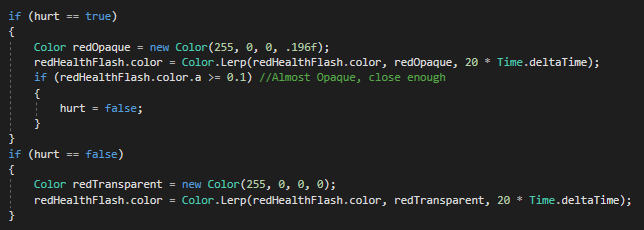
 Any action-based game has some sort of health system. Various in-game interactions will either give or take health from the player. The player health was established by writing a script and creating a float-type variable called *playerHealth.* A parameter was placed on the health value, limiting it to a maximum of 100. Therefore, if a player completed an objective that grants health and that puts it over 100, it will remain at 100. Also, this same script provides visuals for health changes. The screen will flash red when the player takes damage, and it will flash green when the player is healed. This feature was created by using the *Lerp* method to change a red and green panel’s alpha (transparency) value over a short period of time using linear interpolation. These panels are attached to the VR camera and remain completely transparent until the method is called. In order to create a health change, the script causing it will access the *playerHealth* variable in this script, add or subtract from that value, and then call the correct method for the screen flash.

Figure 30. Red screen flash code sample

**Subtask 2. Damage Output**

In the game, the player will frequently find himself fighting off enemies. Obviously, the player will need a weapon to fight back. We decided to create 3 different guns the player can use throughout the game: a pistol, an assault rifle, and a grenade launcher. Each gun has its own damage output values and will be found during different times while playing the game.



Figure 31. Pictures of each of the guns found in the game

The player starts the game with just the pistol. It is the weakest gun in the game for a few reasons. It has a magazine size of 15 shots, the slowest bullet speed, and only does 7 points of damage to enemies. This will be the only weapon the player has until multiple choice question 2. The assault rifle is the reward for getting question 2 correct. The assault rifle has a faster bullet speed and does 10 damage to the enemies. It has a clip size of 30 bullets, allowing the player to shoot more consistently. The final gun found in the game is the grenade launcher. The clip size is unlimited, does enough damage to kill enemies in one hit, and is useful for the final puzzle in the game. However, the gun has a limited amount of shots, with that number being 20. As a hidden bonus for the player, if they answer all multiple choice questions correctly, a total of 1000 ammo will be added to each clip, allowing them to shoot rapidly for the rest of the game.

**Subtask 3. Movement**

In virtual reality, physically moving, whether it's walking, running, crouching, or jumping, is obviously the main movement option since the system tracks real-world movements. However, if the physical movement was the only movement option, the game would be constrained to the size of the room the player is playing in. Fortunately, the SteamVR plugin includes a feature called teleporting. A player can teleport by pressing up on the directional pad on the Vive controller. This allows the player to jump from place to place in-game without physically moving. Having this feature circumvents room space constraints, reduces the potential for physical fatigue in the player, and also provides a tactical movement option in enemy fights.

**A picture containing indoor, next, black

Description automatically generated**

Figure 32. Picture display Vive controller input and player view of teleportation marker

## Implementation of Task 5. Features

The following subtasks involve key pieces to the game that are used or experienced frequently throughout the game. While the features are not objectives that need to be fulfilled to beat the game, they were vital in allowing the game to work and flow correctly.

**Subtask 1. Heads-Up Display (HUD)**

In every video game, the player has important information to keep in mind as he goes through the game, such as health, ammo, score, etc. This information is typically found around the outer parts of the screen. The collection of objects displaying this information is called the heads-up display, or HUD. This game is no different in that aspect since the player has to keep track of health and ammo amounts. Since the first-person view in the VR headset is not ideal for the traditional HUD layout around the screens, we decided to develop a wrist-attached HUD.

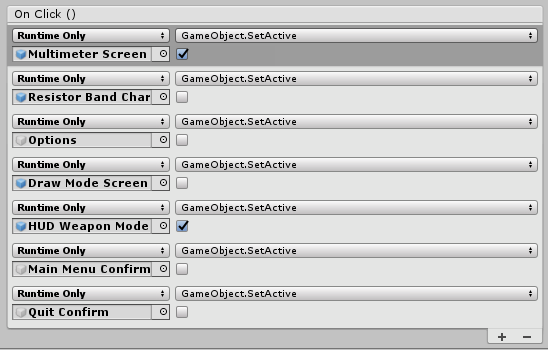
The base of the HUD is a computer tablet found on the asset store (called the SciFi Tablet). The object was attached to the left-hand object in the SteamVR player asset. Since it is attached to the hand itself, it will following the position and rotation of the hand. Now that the base of the object was established, the UI elements could be attached. The HUD has 4 different screens: an options screen, a screen to display a resistor band chart, a screen to display multimeter readings, and a screen for draw mode (feature will be explained in a later section). Four buttons, one for each screen, were placed at the top of the screen. Each button was given a box collider that acts as a trigger. Using the *OnTriggerEnter* method with a “Player” tag check, and we can set the buttons to act as though they are being clicked on by invoking their *OnClick* method. That triggers all events selected in the On Click section for that button in the Unity inspector. The menu buttons will activate their intended screen, and then deactivate all other screens. The HUD also displays ammo counts and health that are viewable on each screen except for the draw mode screen.

Figure 33. View of On Click events from the Unity inspector

The options screens is a simple screen that contains buttons to return to the main menu or to quit out of the game. This is controlled by a single script containing mehods to change the scene loaded in the game

**Subtask 4. Gun System**

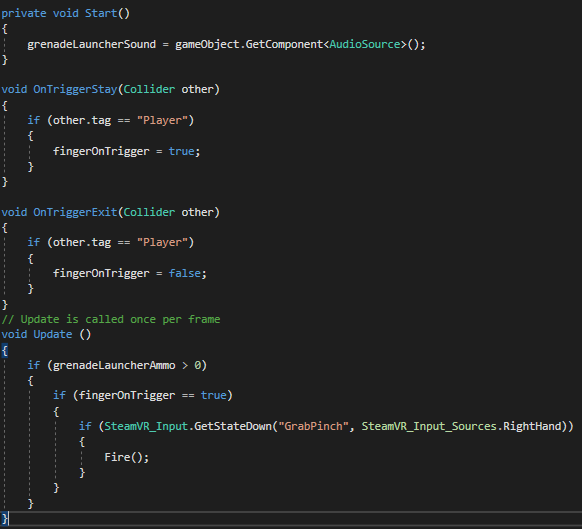
While all guns have their own attributes, designing and scripting their shooting functionality had a lot of cross-over. For each gun, an empty object named “BulletSpawn” had to be created at the very end of the barrel. In each of the scripts for the guns, conditions were created to allow the player to shoot it. First, the script will check if the player is holding the gun so that his finger is on the trigger. This was achieved by creating a box collider on the trigger of the gun and setting it has a trigger. Then, with the script attached to the trigger object, write an *OnTriggerEnter* method that will check for an object with the tag “Player”. In physical terms, this checks if the player’s hand is making contact with the trigger of the gun. If this occurs, the bool variable *fingerOnTrigger* is set to true. The next condition that needs to be met is whether or not there is ammo in the gun. Creating an int variable *ammoInMag,* then checking if that value is greater than 0 solved that problem. The final condition that needs to be met is that the player needs to be pulling the trigger. Using the SteamVR’s *Valve.VR* library, we were able to check the state of the trigger on the Vive controller and use it inside of an if-statement. As a recap, the script must check if the player’s finger is on the trigger, check if there is ammo in the magazine, and check if the trigger on the Vive controller is being pulled.

Figure 34. Condtions to be satisfied to fire any gun (Specific screenshot from Grenade Launcher script)

As shown in the code sample above, once all the conditions are met, the method *Fire* is set to run. This method contains a set of instructions that cause the gun to fire. First, a bullet object is chosen, then entered as a public variable. It is created using the *Instantiate* method. The bullet is instantiated at the position and rotation values of the previously mentioned “BulletSpawn” object. Next, the script accesses the “Rigid Body” component of the bullet and sends it at a constant velocity, which is controlled by another public variable called *bulletSpeed*. Now that the physics of the bullet was established, the visual and audio effects can be implemented. The two effects used were a gunshot audio clip and a muzzle flash. For the gunshot sound, an “Audio Source” component was attached to the trigger object along with the script and the desired gunshot sound was placed in the “Audio Clip” box in the component. The script accesses the audio source using the *GetComponent* method and assigns it to a private variable. The *Play* method is then called for the variable, playing the sound. The muzzle flash is exclusively for the assault rifle and pistol. The muzzle flash objects are “Particle Systems” and are attached to the gun at the “BulletSpawn” position. It is then accessed by using the *GetComponent* method once again and then displayed using the *Play* method. To finish the script, an instruction to subtract 1 from *ammoInMag* was written so that every time the gun is fired, a bullet is lost from the magazine.

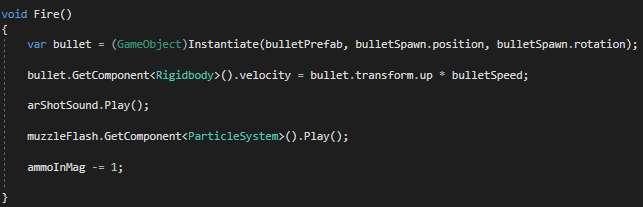


Figure 35. Code sample from AssaultRifle script, *Fire* method

1. Conclusion.

At the conclusion of the first semester working on this project, I have learned a lot about the design process in game development. The main issue I have encountered is not having a well-defined design method in place. For the second semester, I will be sure to have a stronger approach for laying out objectives, so I can achieve the goals faster and more efficiently.

In terms of the technical information I have gained, I learned just about everything about C# programming that I know from this project. This is providing me with a great foundation in object-based programming that can be used in many programming professions. I’ve gained more from this project so far than most classes throughout my entire time in the Electric Engineering program here at UMES. Overall, this project has undoubtedly made me a better engineer in just one semester.

Acknowledgment

Joe McGinley - one of my best friends, and a brilliant programmer who’s worked with Unity and helped me along with the way.

Appendix

1. Script: AlienCollisionDetection

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class AlienCollisionDetection : MonoBehaviour

{

void OnCollisionEnter(Collision collision)

{

if(collision.gameObject.tag == "Pistol Bullet")

{

var pistolshothit = GetComponentInParent<RangedEnemy>().health -= 5.0f;

}

if(collision.gameObject.tag == "AR Bullet")

{

var ARShotHit = GetComponentInParent<RangedEnemy>().health -= 9.0f;

}

if(collision.gameObject.tag == "Grenade")

{

var grenadeShotHit = GetComponentInParent<RangedEnemy>().health -= 70.0f;

}

}

}

1. Script: CircuitOpensDoor

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CircuitOpensDoor : MonoBehaviour

{

//This is the script used to create a voltage divider circuit in the game for one of the circuit design objectives in the game.

public GameObject inputPort; //Choose either bottom or top piece of the port as your port for both input and output, I chose bottom for both input and output

public GameObject outputPort;

public GameObject batteryCase; //Choose either terminal as a distance reference, I choose the negative terminal

private Vector3 inputLocation; //Locations of the input, output, and battery port to find distance between

private Vector3 outputLocation;

private Vector3 batteryLocation;

public GameObject resistor5000; //Resistor objects with their assigned resistances

public GameObject resistor4000;

public GameObject resistor3000;

public GameObject resistor2500;

public GameObject resistor1500;

public GameObject resistor500;

public GameObject battery90; //Battery objects with their assigned voltages

public GameObject battery120;

public GameObject battery150;

private float inputResistance = 0.0f; //Float variable for resistance of the resistor plugged into the input resistor port

private float outputResistance = 0.0f; //Float variable for resistance of the resistor plugged into the output resistor port

private float inputVoltage = 0.0f; //Float variable for voltage of battery inserted into the port

public float outputVoltage = 0.0f; //Calculation of voltage using voltage divider formula

public float outputResistorPower = 0.0f; //Calulation of power dissipated by the output resistor using the power formula;

public bool inputResistorInserted = false; //Bool values tracking whether a resistor/battery is inserted into the respective ports

public bool outputResistorInserted = false;

public bool batteryInserted = false;

public Animator doorAnimator;

public Renderer doorLight;

public Material baseColor;

public Material greenEmission;

public Material[] correctCircuitLight;

public void Start()

{

inputLocation = inputPort.transform.position;

outputLocation = outputPort.transform.position;

batteryLocation = batteryCase.transform.position;

correctCircuitLight[0] = baseColor;

correctCircuitLight[1] = greenEmission;

}

public void Update()

{

//Input resistor arguments

if (Vector3.Distance(resistor5000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 5000.0f;

}

else if (Vector3.Distance(resistor4000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 4000.0f;

}

else if (Vector3.Distance(resistor3000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 3000.0f;

}

else if (Vector3.Distance(resistor2500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 2500.0f;

}

else if (Vector3.Distance(resistor1500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 1500.0f;

}

else if (Vector3.Distance(resistor500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 500.0f;

}

else

{

inputResistorInserted = false;

inputResistance = 0.0f;

}

//Ouptut resistor arguments

if (Vector3.Distance(resistor5000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 5000.0f;

}

else if (Vector3.Distance(resistor4000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 4000.0f;

}

else if (Vector3.Distance(resistor3000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 3000.0f;

}

else if (Vector3.Distance(resistor2500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 2500.0f;

}

else if (Vector3.Distance(resistor1500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 1500.0f;

}

else if (Vector3.Distance(resistor500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 500.0f;

}

else

{

outputResistorInserted = false;

outputResistance = 0.0f;

}

//Battery arguments

if (Vector3.Distance(battery90.transform.position, batteryLocation) < 0.5f)

{

batteryInserted = true;

inputVoltage = 90.0f;

}

//Voltage Divider and Power Equations

outputVoltage = (inputVoltage \* outputResistance) / (inputResistance + outputResistance);

outputResistorPower = (outputVoltage \* outputVoltage) / outputResistance;

if (outputVoltage == 60.0f)

{

doorLight.materials = correctCircuitLight;

if (this.gameObject.tag == "On")

{

doorAnimator.SetTrigger("character\_nearby");

}

}

}

1. Script: OptionButtonPress

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class OptionButtonPress : MonoBehaviour

{

public GameObject resistorBandChart;

public GameObject multimeter;

public GameObject Notepad;

public GameObject HUDWeaponModels;

public GameObject resBandButton;

public GameObject multimeterButton;

public GameObject notepadButton;

public bool test;

void OnTriggerEnter(Collider col)

{

if (col.gameObject.tag == "Player")

{

resistorBandChart.SetActive(false);

multimeter.SetActive(false);

Notepad.SetActive(false);

HUDWeaponModels.SetActive(true);

PressedButtonColor();

ResetOtherButtonColors();

test = true;

}

}

void PressedButtonColor()

{

var pressedColor = new Color32(255, 182, 182, 255);

var thisbuttonColor = this.gameObject.GetComponent<Image>().color;

thisbuttonColor = pressedColor;

this.gameObject.GetComponent<Image>().color = thisbuttonColor;

}

void ResetOtherButtonColors()

{

var originalbuttoncolor = new Color32(96, 9, 9, 255);

var resbandbuttcolor = resBandButton.GetComponent<Image>().color;

var multimeterbuttcolor = multimeterButton.GetComponent<Image>().color;

var notepadbuttcolor = notepadButton.GetComponent<Image>().color;

resbandbuttcolor = originalbuttoncolor;

multimeterbuttcolor = originalbuttoncolor;

notepadbuttcolor = originalbuttoncolor;

resBandButton.GetComponent<Image>().color = resbandbuttcolor;

multimeterButton.GetComponent<Image>().color = multimeterbuttcolor;

notepadButton.GetComponent<Image>().color = notepadbuttcolor;

}

1. Script for Reloading (Assault Rifle)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using Valve.VR;

public class ARReload : MonoBehaviour

{

public GameObject assaultRifleTrigger;

public bool reloadOnce;

public bool leftHandOnMag;

public bool clipInHand;

private AudioSource reloadSound;

void Start()

{

reloadSound = gameObject.GetComponent<AudioSource>();

}

void OnTriggerEnter(Collider other)

{

if (other.transform.gameObject.name == "LeftHand")

{

leftHandOnMag = true;

}

}

void OnTriggerExit(Collider other)

{

if (other.transform.gameObject.name == "LeftHand")

{

leftHandOnMag = false;

reloadOnce = false;

}

}

public void Update()

{

if ((SteamVR\_Input.GetStateDown("GrabGrip", SteamVR\_Input\_Sources.LeftHand)))

{

clipInHand = true;

}

if (reloadOnce == false)

{

if (clipInHand == true)

{

if (leftHandOnMag == true)

{

Reload();

reloadOnce = true;

clipInHand = false;

}

}

}

}

public void Reload()

{

reloadSound.Play();

var reload = assaultRifleTrigger.GetComponent<AssaultRifle>().ammoInMag = 30;

}

}

1. Gun Script (Assault Rifle Version)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using Valve.VR;

public class AssaultRifle : MonoBehaviour

{

public Transform bulletSpawn;

public GameObject bulletPrefab;

public GameObject muzzleFlash;

public bool fingerOnTrigger = false;

public float bulletSpeed;

private AudioSource arShotSound;

public int ammoInMag;

public Text arAmmoText;

private void Start()

{

arShotSound = gameObject.GetComponent<AudioSource>();

}

void OnTriggerStay(Collider other)

{

if (other.tag == "Player")

{

fingerOnTrigger = true;

}

}

void OnTriggerExit(Collider other)

{

if (other.tag == "Player")

{

fingerOnTrigger = false;

}

}

// Update is called once per frame

void Update()

{

arAmmoText.text = ammoInMag.ToString();

if (ammoInMag > 0)

{

if (fingerOnTrigger == true)

{

if (SteamVR\_Input.GetStateDown("GrabPinch", SteamVR\_Input\_Sources.RightHand))

{

Fire();

}

}

}

}

void Fire()

{

var bullet = (GameObject)Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);

bullet.GetComponent<Rigidbody>().velocity = bullet.transform.up \* bulletSpeed;

arShotSound.Play();

muzzleFlash.GetComponent<ParticleSystem>().Play();

ammoInMag -= 1;

}

}

1. Script for Opening Door by Completing Circuit

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CircuitOpensDoor : MonoBehaviour

{

//This is the script used to create a voltage divider circuit in the game for one of the circuit design objectives in the game.

public GameObject inputPort; //Choose either bottom or top piece of the port as your port for both input and output, I chose bottom for both input and output

public GameObject outputPort;

public GameObject batteryCase; //Choose either terminal as a distance reference, I choose the negative terminal

private Vector3 inputLocation; //Locations of the input, output, and battery port to find distance between

private Vector3 outputLocation;

private Vector3 batteryLocation;

public GameObject resistor5000; //Resistor objects with their assigned resistances

public GameObject resistor4000;

public GameObject resistor3000;

public GameObject resistor2500;

public GameObject resistor1500;

public GameObject resistor500;

public GameObject battery90; //Battery objects with their assigned voltages

public GameObject battery120;

public GameObject battery150;

private float inputResistance = 0.0f; //Float variable for resistance of the resistor plugged into the input resistor port

private float outputResistance = 0.0f; //Float variable for resistance of the resistor plugged into the output resistor port

private float inputVoltage = 0.0f; //Float variable for voltage of battery inserted into the port

public float outputVoltage = 0.0f; //Calculation of voltage using voltage divider formula

public float outputResistorPower = 0.0f; //Calulation of power dissipated by the output resistor using the power formula;

public bool inputResistorInserted = false; //Bool values tracking whether a resistor/battery is inserted into the respective ports

public bool outputResistorInserted = false;

public bool batteryInserted = false;

public Animator doorAnimator;

public Renderer doorLight;

public Material baseColor;

public Material greenEmission;

public Material[] correctCircuitLight;

public void Start()

{

inputLocation = inputPort.transform.position;

outputLocation = outputPort.transform.position;

batteryLocation = batteryCase.transform.position;

correctCircuitLight[0] = baseColor;

correctCircuitLight[1] = greenEmission;

}

public void Update()

{

//Input resistor arguments

if (Vector3.Distance(resistor5000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 5000.0f;

}

else if (Vector3.Distance(resistor4000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 4000.0f;

}

else if (Vector3.Distance(resistor3000.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 3000.0f;

}

else if (Vector3.Distance(resistor2500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 2500.0f;

}

else if (Vector3.Distance(resistor1500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 1500.0f;

}

else if (Vector3.Distance(resistor500.transform.position, inputLocation) < 0.8f)

{

inputResistorInserted = true;

inputResistance = 500.0f;

}

else

{

inputResistorInserted = false;

inputResistance = 0.0f;

}

//Ouptut resistor arguments

if (Vector3.Distance(resistor5000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 5000.0f;

}

else if (Vector3.Distance(resistor4000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 4000.0f;

}

else if (Vector3.Distance(resistor3000.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 3000.0f;

}

else if (Vector3.Distance(resistor2500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 2500.0f;

}

else if (Vector3.Distance(resistor1500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 1500.0f;

}

else if (Vector3.Distance(resistor500.transform.position, outputLocation) < 0.8f)

{

outputResistorInserted = true;

outputResistance = 500.0f;

}

else

{

outputResistorInserted = false;

outputResistance = 0.0f;

}

//Battery arguments

if (Vector3.Distance(battery90.transform.position, batteryLocation) < 0.5f)

{

batteryInserted = true;

inputVoltage = 90.0f;

}

//Voltage Divider and Power Equations

outputVoltage = (inputVoltage \* outputResistance) / (inputResistance + outputResistance);

outputResistorPower = (outputVoltage \* outputVoltage) / outputResistance;

if (outputVoltage == 60.0f)

{

doorLight.materials = correctCircuitLight;

if (this.gameObject.tag == "On")

{

doorAnimator.SetTrigger("character\_nearby");

}

}

}

}

1. Script for HUD Button Presses (Draw Mode Button)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class DrawModeButton : MonoBehaviour

{

public GameObject clockButton;

public GameObject resBandButton;

public GameObject multimeterButton;

public bool test;

void OnTriggerEnter(Collider col)

{

if (col.gameObject.tag == "Player")

{

gameObject.GetComponent<Button>().onClick.Invoke();

PressedButtonColor();

ResetOtherButtonColors();

}

}

void PressedButtonColor()

{

var pressedColor = new Color32(255, 182, 182, 255);

var thisbuttonColor = this.gameObject.GetComponent<Image>().color;

thisbuttonColor = pressedColor;

this.gameObject.GetComponent<Image>().color = thisbuttonColor;

}

void ResetOtherButtonColors()

{

var originalbuttoncolor = new Color32(96, 9, 9, 255);

var resbandbuttcolor = resBandButton.GetComponent<Image>().color;

var clockbuttcolor = clockButton.GetComponent<Image>().color;

var multimeterbuttcolor = multimeterButton.GetComponent<Image>().color;

resbandbuttcolor = originalbuttoncolor;

clockbuttcolor = originalbuttoncolor;

multimeterbuttcolor = originalbuttoncolor;

resBandButton.GetComponent<Image>().color = resbandbuttcolor;

clockButton.GetComponent<Image>().color = clockbuttcolor;

multimeterButton.GetComponent<Image>().color = multimeterbuttcolor;

}

}

1. Script for Drawing in 3-D Space

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using Valve.VR;

public class DrawingScript : MonoBehaviour

{

private GameObject ink;

public float drawRate;

private float lastDrawn;

public bool drawModeOn = false;

public GameObject redInk;

public GameObject orangeInk;

public GameObject yellowInk;

public GameObject greenInk;

public GameObject blueInk;

public GameObject purpleInk;

public bool redOn = true;

public bool orangeOn = false;

public bool yellowOn = false;

public bool greenOn = false;

public bool blueOn = false;

public bool purpleOn = false;

public bool eraserOn = false;

private void OnTriggerEnter(Collider other)

{

if (other.gameObject.tag == "Drawn")

{

if (drawModeOn == true)

{

if(eraserOn == true)

{

if (SteamVR\_Input.GetState("GrabPinch", SteamVR\_Input\_Sources.RightHand))

{

Destroy(other);

}

}

}

}

}

void Update()

{

if(redOn == true)

{

ink = redInk;

}

if(orangeOn == true)

{

ink = orangeInk;

}

if (yellowOn == true)

{

ink = yellowInk;

}

if (greenOn == true)

{

ink = greenInk;

}

if (blueOn == true)

{

ink = blueInk;

}

if (purpleOn == true)

{

ink = purpleInk;

}

if(eraserOn == false)

{

if (drawModeOn == true)

{

if (SteamVR\_Input.GetState("GrabPinch", SteamVR\_Input\_Sources.RightHand))

{

if (Time.time - lastDrawn > 1 / drawRate)

{

lastDrawn = Time.time;

Instantiate(ink, this.gameObject.transform.position, this.gameObject.transform.rotation);

}

}

}

}

}

}

1. Script for timed enemy spawns

using UnityEngine;

public class EnemySpawn : MonoBehaviour

{

public GameObject enemy; // The enemy prefab to be spawned.

public GameObject enemyParent; // Parent object used to keep count of total enemies on map

public GameObject spawnEffect; // Particle system for the spawn effect under the enemies

public float spawnTime; // How long between each spawn.

public Transform[] spawnPoints; // An array of the spawn points this enemy can spawn from.

public AudioClip spawnSound;

void Start()

{

// Call the Spawn function after a delay of the spawnTime and then continue to call after the same amount of time.

InvokeRepeating("Spawn", spawnTime, spawnTime);

}

void Spawn()

{

// Find a random index between zero and one less than the number of spawn points.

int spawnPointIndex = Random.Range(0, spawnPoints.Length);

// Create an instance of the enemy prefab at the randomly selected spawn point's position and rotation.

Instantiate(enemy, spawnPoints[spawnPointIndex].position, spawnPoints[spawnPointIndex].rotation);

var spawnEffectInstant = Instantiate(spawnEffect, spawnPoints[spawnPointIndex].position, spawnPoints[spawnPointIndex].rotation);

AudioSource.PlayClipAtPoint(spawnSound, spawnPoints[spawnPointIndex].position);

Destroy(spawnEffectInstant, 3.3f);

}

}

1. Script for floating effect on key projections

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class FloatingProjections : MonoBehaviour

{

public float speedMultiplier;

public float amplitude = 0.5f;

public float frequency = 1f;

Vector3 posOffset = new Vector3();

Vector3 tempPos = new Vector3();

void Start()

{

posOffset = transform.position;

}

void Update()

{

this.gameObject.transform.Rotate(0, speedMultiplier \* Time.deltaTime, 0);

tempPos = posOffset;

tempPos.y += Mathf.Sin(Time.fixedTime \* Mathf.PI \* frequency) \* amplitude;

transform.position = tempPos;

}

}

1. Script for player health

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class Health : MonoBehaviour

{

public float playerHealth;

public float maxHealth;

public Text healthText;

public GameObject redHealthFlashObject;

private Image redHealthFlash;

public GameObject greenHealthFlashObject;

private Image greenHealthFlash;

private float targetAlpha = 1;

private float fadeDuration = .05f;

private float lerpParam;

private float startAlpha = 0;

public bool hurt;

public bool healed;

void Start()

{

playerHealth = maxHealth;

redHealthFlash = redHealthFlashObject.GetComponent<Image>();

greenHealthFlash = greenHealthFlashObject.GetComponent<Image>();

}

void Update()

{

healthText.text = "Health: " + playerHealth.ToString(); //Health displayed on HUD

if (playerHealth > maxHealth)

{

playerHealth = 100.0f; //Caps health at 100

}

if (hurt == true)

{

Color redOpaque = new Color(255, 0, 0, .196f);

redHealthFlash.color = Color.Lerp(redHealthFlash.color, redOpaque, 20 \* Time.deltaTime);

if (redHealthFlash.color.a >= 0.1) //Almost Opaque, close enough

{

hurt = false;

}

}

if (hurt == false)

{

Color redTransparent = new Color(255, 0, 0, 0);

redHealthFlash.color = Color.Lerp(redHealthFlash.color, redTransparent, 20 \* Time.deltaTime);

}

if (healed == true)

{

Color greenOpaque = new Color(0, 255, 0, .196f);

greenHealthFlash.color = Color.Lerp(greenHealthFlash.color, greenOpaque, 5 \* Time.deltaTime);

if (greenHealthFlash.color.a >= 0.1) //Almost Opaque, close enough

{

healed = false;

}

}

if (healed == false)

{

Color greenTransparent = new Color(0, 255, 0, 0);

greenHealthFlash.color = Color.Lerp(greenHealthFlash.color, greenTransparent, 5 \* Time.deltaTime);

}

}

}

1. Script controlling HUD Visibility

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class Health : MonoBehaviour

{

public float playerHealth;

public float maxHealth;

public Text healthText;

public GameObject redHealthFlashObject;

private Image redHealthFlash;

public GameObject greenHealthFlashObject;

private Image greenHealthFlash;

private float targetAlpha = 1;

private float fadeDuration = .05f;

private float lerpParam;

private float startAlpha = 0;

public bool hurt;

public bool healed;

void Start()

{

playerHealth = maxHealth;

redHealthFlash = redHealthFlashObject.GetComponent<Image>();

greenHealthFlash = greenHealthFlashObject.GetComponent<Image>();

}

void Update()

{

healthText.text = "Health: " + playerHealth.ToString(); //Health displayed on HUD

if (playerHealth > maxHealth)

{

playerHealth = 100.0f; //Caps health at 100

}

if (hurt == true)

{

Color redOpaque = new Color(255, 0, 0, .196f);

redHealthFlash.color = Color.Lerp(redHealthFlash.color, redOpaque, 20 \* Time.deltaTime);

if (redHealthFlash.color.a >= 0.1) //Almost Opaque, close enough

{

hurt = false;

}

}

if (hurt == false)

{

Color redTransparent = new Color(255, 0, 0, 0);

redHealthFlash.color = Color.Lerp(redHealthFlash.color, redTransparent, 20 \* Time.deltaTime);

}

if (healed == true)

{

Color greenOpaque = new Color(0, 255, 0, .196f);

greenHealthFlash.color = Color.Lerp(greenHealthFlash.color, greenOpaque, 5 \* Time.deltaTime);

if (greenHealthFlash.color.a >= 0.1) //Almost Opaque, close enough

{

healed = false;

}

}

if (healed == false)

{

Color greenTransparent = new Color(0, 255, 0, 0);

greenHealthFlash.color = Color.Lerp(greenHealthFlash.color, greenTransparent, 5 \* Time.deltaTime);

}

}

}

1. Script controlling colors of selected multiple choice question

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

//This script will control the color of the on screen buttons so the player has a visual of what answer was selected. They will shift to a brighter color of the default color.

//This is attached to mult. choice canvas of the desired screen.

public class MultChoiceButtonColorControl : MonoBehaviour

{

public GameObject onScreenAButton; //Public variable for each of the 4 buttons on the screen

public GameObject onScreenBButton;

public GameObject onScreenCButton;

public GameObject onScreenDButton;

private Color originalButtonColor; //The color of the button when not selected

private Color selectedButtonColor; //The desired color when the corresponding answer is chosen

// Start is called before the first frame update

void Start()

{

originalButtonColor = onScreenAButton.GetComponent<Image>().color; //Attaches color values to each of the variables at start up

selectedButtonColor = new Color32(255, 182, 182, 255);

}

// Update is called once per frame

void Update()

{

if(this.gameObject.tag == "A") //If A is selected, the on screen A button will change colors

{

onScreenAButton.GetComponent<Image>().color = selectedButtonColor;

onScreenBButton.GetComponent<Image>().color = originalButtonColor;

onScreenCButton.GetComponent<Image>().color = originalButtonColor;

onScreenDButton.GetComponent<Image>().color = originalButtonColor;

}

if (this.gameObject.tag == "B") //If B is selected, the on screen B button will change colors

{

onScreenAButton.GetComponent<Image>().color = originalButtonColor;

onScreenBButton.GetComponent<Image>().color = selectedButtonColor;

onScreenCButton.GetComponent<Image>().color = originalButtonColor;

onScreenDButton.GetComponent<Image>().color = originalButtonColor;

}

if (this.gameObject.tag == "C") //"" C "" C Button

{

onScreenAButton.GetComponent<Image>().color = originalButtonColor;

onScreenBButton.GetComponent<Image>().color = originalButtonColor;

onScreenCButton.GetComponent<Image>().color = selectedButtonColor;

onScreenDButton.GetComponent<Image>().color = originalButtonColor;

}

if (this.gameObject.tag == "D") //"" D "" D Button

{

onScreenAButton.GetComponent<Image>().color = originalButtonColor;

onScreenBButton.GetComponent<Image>().color = originalButtonColor;

onScreenCButton.GetComponent<Image>().color = originalButtonColor;

onScreenDButton.GetComponent<Image>().color = selectedButtonColor;

}

}

}

1. Script for multimeter built into HUD

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class MultimeterScript : MonoBehaviour

{

public string measuredResistance;

private string measuredVoltage;

public Text multimeterReading;

void OnTriggerEnter(Collider other)

{

if (other.gameObject.tag == "Untagged")

{

multimeterReading.text = "Insert Resistor";

}

measuredResistance = other.gameObject.tag;

multimeterReading.text = measuredResistance;

}

private void OnTriggerExit(Collider other)

{

multimeterReading.text = "Insert Resistor";

}

}

1. Script controlling background music
2. Script for player hit box/taking damage
3. Script for setting question data to playerprefs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class PlayerPrefMultChoice : MonoBehaviour

{

public Text q1Option1Text;

public Text q1Option1ChoiceA;

public Text q1Option1ChoiceB;

public Text q1Option1ChoiceC;

public Text q1Option1ChoiceD;

public Text q1Option1CorrectAnswer;

public Text q1Option2Text;

public Text q1Option2ChoiceA;

public Text q1Option2ChoiceB;

public Text q1Option2ChoiceC;

public Text q1Option2ChoiceD;

public Text q1Option2CorrectAnswer;

public Text q1Option3Text;

public Text q1Option3ChoiceA;

public Text q1Option3ChoiceB;

public Text q1Option3ChoiceC;

public Text q1Option3ChoiceD;

public Text q1Option3CorrectAnswer;

public Text q2Option1Text;

public Text q2Option1ChoiceA;

public Text q2Option1ChoiceB;

public Text q2Option1ChoiceC;

public Text q2Option1ChoiceD;

public Text q2Option1CorrectAnswer;

public Text q2Option2Text;

public Text q2Option2ChoiceA;

public Text q2Option2ChoiceB;

public Text q2Option2ChoiceC;

public Text q2Option2ChoiceD;

public Text q2Option2CorrectAnswer;

public Text q2Option3Text;

public Text q2Option3ChoiceA;

public Text q2Option3ChoiceB;

public Text q2Option3ChoiceC;

public Text q2Option3ChoiceD;

public Text q2Option3CorrectAnswer;

public Text q3Option1Text;

public Text q3Option1ChoiceA;

public Text q3Option1ChoiceB;

public Text q3Option1ChoiceC;

public Text q3Option1ChoiceD;

public Text q3Option1CorrectAnswer;

public Text q3Option2Text;

public Text q3Option2ChoiceA;

public Text q3Option2ChoiceB;

public Text q3Option2ChoiceC;

public Text q3Option2ChoiceD;

public Text q3Option2CorrectAnswer;

public Text q3Option3Text;

public Text q3Option3ChoiceA;

public Text q3Option3ChoiceB;

public Text q3Option3ChoiceC;

public Text q3Option3ChoiceD;

public Text q3Option3CorrectAnswer;

public Text q4Option1Text;

public Text q4Option1ChoiceA;

public Text q4Option1ChoiceB;

public Text q4Option1ChoiceC;

public Text q4Option1ChoiceD;

public Text q4Option1CorrectAnswer;

public Text q4Option2Text;

public Text q4Option2ChoiceA;

public Text q4Option2ChoiceB;

public Text q4Option2ChoiceC;

public Text q4Option2ChoiceD;

public Text q4Option2CorrectAnswer;

public Text q4Option3Text;

public Text q4Option3ChoiceA;

public Text q4Option3ChoiceB;

public Text q4Option3ChoiceC;

public Text q4Option3ChoiceD;

public Text q4Option3CorrectAnswer;

public Text q5Option1Text;

public Text q5Option1ChoiceA;

public Text q5Option1ChoiceB;

public Text q5Option1ChoiceC;

public Text q5Option1ChoiceD;

public Text q5Option1CorrectAnswer;

public Text q5Option2Text;

public Text q5Option2ChoiceA;

public Text q5Option2ChoiceB;

public Text q5Option2ChoiceC;

public Text q5Option2ChoiceD;

public Text q5Option2CorrectAnswer;

public Text q5Option3Text;

public Text q5Option3ChoiceA;

public Text q5Option3ChoiceB;

public Text q5Option3ChoiceC;

public Text q5Option3ChoiceD;

public Text q5Option3CorrectAnswer;

void Update()

{

PlayerPrefs.SetString("Question 1 Option 1 Text", q1Option1Text.text);

PlayerPrefs.SetString("Question 1 Option 1 Choice A", q1Option1ChoiceA.text);

PlayerPrefs.SetString("Question 1 Option 1 Choice B", q1Option1ChoiceB.text);

PlayerPrefs.SetString("Question 1 Option 1 Choice C", q1Option1ChoiceC.text);

PlayerPrefs.SetString("Question 1 Option 1 Choice D", q1Option1ChoiceD.text);

PlayerPrefs.SetString("Question 1 Option 1 Correct Answer", q1Option1CorrectAnswer.text);

PlayerPrefs.SetString("Question 1 Option 2 Text", q1Option2Text.text);

PlayerPrefs.SetString("Question 1 Option 2 Choice A", q1Option2ChoiceA.text);

PlayerPrefs.SetString("Question 1 Option 2 Choice B", q1Option2ChoiceB.text);

PlayerPrefs.SetString("Question 1 Option 2 Choice C", q1Option2ChoiceC.text);

PlayerPrefs.SetString("Question 1 Option 2 Choice D", q1Option2ChoiceD.text);

PlayerPrefs.SetString("Question 1 Option 2 Correct Answer", q1Option2CorrectAnswer.text);

PlayerPrefs.SetString("Question 1 Option 3 Text", q1Option3Text.text);

PlayerPrefs.SetString("Question 1 Option 3 Choice A", q1Option3ChoiceA.text);

PlayerPrefs.SetString("Question 1 Option 3 Choice B", q1Option3ChoiceB.text);

PlayerPrefs.SetString("Question 1 Option 3 Choice C", q1Option3ChoiceC.text);

PlayerPrefs.SetString("Question 1 Option 3 Choice D", q1Option3ChoiceD.text);

PlayerPrefs.SetString("Question 1 Option 3 Correct Answer", q1Option3CorrectAnswer.text);

PlayerPrefs.SetString("Question 2 Option 1 Text", q2Option1Text.text);

PlayerPrefs.SetString("Question 2 Option 1 Choice A", q2Option1ChoiceA.text);

PlayerPrefs.SetString("Question 2 Option 1 Choice B", q2Option1ChoiceB.text);

PlayerPrefs.SetString("Question 2 Option 1 Choice C", q2Option1ChoiceC.text);

PlayerPrefs.SetString("Question 2 Option 1 Choice D", q2Option1ChoiceD.text);

PlayerPrefs.SetString("Question 2 Option 1 Correct Answer", q2Option1CorrectAnswer.text);

PlayerPrefs.SetString("Question 2 Option 2 Text", q1Option2Text.text);

PlayerPrefs.SetString("Question 2 Option 2 Choice A", q2Option2ChoiceA.text);

PlayerPrefs.SetString("Question 2 Option 2 Choice B", q2Option2ChoiceB.text);

PlayerPrefs.SetString("Question 2 Option 2 Choice C", q2Option2ChoiceC.text);

PlayerPrefs.SetString("Question 2 Option 2 Choice D", q2Option2ChoiceD.text);

PlayerPrefs.SetString("Question 2 Option 2 Correct Answer", q2Option2CorrectAnswer.text);

PlayerPrefs.SetString("Question 2 Option 3 Text", q2Option3Text.text);

PlayerPrefs.SetString("Question 2 Option 3 Choice A", q2Option3ChoiceA.text);

PlayerPrefs.SetString("Question 2 Option 3 Choice B", q2Option3ChoiceB.text);

PlayerPrefs.SetString("Question 2 Option 3 Choice C", q2Option3ChoiceC.text);

PlayerPrefs.SetString("Question 2 Option 3 Choice D", q2Option3ChoiceD.text);

PlayerPrefs.SetString("Question 2 Option 3 Correct Answer", q2Option3CorrectAnswer.text);

PlayerPrefs.SetString("Question 3 Option 1 Text", q3Option1Text.text);

PlayerPrefs.SetString("Question 3 Option 1 Choice A", q3Option1ChoiceA.text);

PlayerPrefs.SetString("Question 3 Option 1 Choice B", q3Option1ChoiceB.text);

PlayerPrefs.SetString("Question 3 Option 1 Choice C", q3Option1ChoiceC.text);

PlayerPrefs.SetString("Question 3 Option 1 Choice D", q3Option1ChoiceD.text);

PlayerPrefs.SetString("Question 3 Option 1 Correct Answer", q3Option1CorrectAnswer.text);

PlayerPrefs.SetString("Question 3 Option 2 Text", q3Option2Text.text);

PlayerPrefs.SetString("Question 3 Option 2 Choice A", q3Option2ChoiceA.text);

PlayerPrefs.SetString("Question 3 Option 2 Choice B", q3Option2ChoiceB.text);

PlayerPrefs.SetString("Question 3 Option 2 Choice C", q3Option2ChoiceC.text);

PlayerPrefs.SetString("Question 3 Option 2 Choice D", q3Option2ChoiceD.text);

PlayerPrefs.SetString("Question 3 Option 2 Correct Answer", q3Option2CorrectAnswer.text);

PlayerPrefs.SetString("Question 3 Option 3 Text", q3Option3Text.text);

PlayerPrefs.SetString("Question 3 Option 3 Choice A", q3Option3ChoiceA.text);

PlayerPrefs.SetString("Question 3 Option 3 Choice B", q3Option3ChoiceB.text);

PlayerPrefs.SetString("Question 3 Option 3 Choice C", q3Option3ChoiceC.text);

PlayerPrefs.SetString("Question 3 Option 3 Choice D", q3Option3ChoiceD.text);

PlayerPrefs.SetString("Question 3 Option 3 Correct Answer", q3Option3CorrectAnswer.text);

PlayerPrefs.SetString("Question 4 Option 1 Text", q4Option1Text.text);

PlayerPrefs.SetString("Question 4 Option 1 Choice A", q4Option1ChoiceA.text);

PlayerPrefs.SetString("Question 4 Option 1 Choice B", q4Option1ChoiceB.text);

PlayerPrefs.SetString("Question 4 Option 1 Choice C", q4Option1ChoiceC.text);

PlayerPrefs.SetString("Question 4 Option 1 Choice D", q4Option1ChoiceD.text);

PlayerPrefs.SetString("Question 4 Option 1 Correct Answer", q4Option1CorrectAnswer.text);

PlayerPrefs.SetString("Question 4 Option 2 Text", q4Option2Text.text);

PlayerPrefs.SetString("Question 4 Option 2 Choice A", q4Option2ChoiceA.text);

PlayerPrefs.SetString("Question 4 Option 2 Choice B", q4Option2ChoiceB.text);

PlayerPrefs.SetString("Question 4 Option 2 Choice C", q4Option2ChoiceC.text);

PlayerPrefs.SetString("Question 4 Option 2 Choice D", q4Option2ChoiceD.text);

PlayerPrefs.SetString("Question 4 Option 2 Correct Answer", q4Option2CorrectAnswer.text);

PlayerPrefs.SetString("Question 4 Option 3 Text", q4Option3Text.text);

PlayerPrefs.SetString("Question 4 Option 3 Choice A", q4Option3ChoiceA.text);

PlayerPrefs.SetString("Question 4 Option 3 Choice B", q4Option3ChoiceB.text);

PlayerPrefs.SetString("Question 4 Option 3 Choice C", q4Option3ChoiceC.text);

PlayerPrefs.SetString("Question 4 Option 3 Choice D", q4Option3ChoiceD.text);

PlayerPrefs.SetString("Question 4 Option 3 Correct Answer", q4Option3CorrectAnswer.text);

PlayerPrefs.SetString("Question 5 Option 1 Text", q5Option1Text.text);

PlayerPrefs.SetString("Question 5 Option 1 Choice A", q5Option1ChoiceA.text);

PlayerPrefs.SetString("Question 5 Option 1 Choice B", q5Option1ChoiceB.text);

PlayerPrefs.SetString("Question 5 Option 1 Choice C", q5Option1ChoiceC.text);

PlayerPrefs.SetString("Question 5 Option 1 Choice D", q5Option1ChoiceD.text);

PlayerPrefs.SetString("Question 5 Option 1 Correct Answer", q5Option1CorrectAnswer.text);

PlayerPrefs.SetString("Question 5 Option 2 Text", q5Option2Text.text);

PlayerPrefs.SetString("Question 5 Option 2 Choice A", q5Option2ChoiceA.text);

PlayerPrefs.SetString("Question 5 Option 2 Choice B", q5Option2ChoiceB.text);

PlayerPrefs.SetString("Question 5 Option 2 Choice C", q5Option2ChoiceC.text);

PlayerPrefs.SetString("Question 5 Option 2 Choice D", q5Option2ChoiceD.text);

PlayerPrefs.SetString("Question 5 Option 2 Correct Answer", q5Option2CorrectAnswer.text);

PlayerPrefs.SetString("Question 5 Option 3 Text", q5Option3Text.text);

PlayerPrefs.SetString("Question 5 Option 3 Choice A", q5Option3ChoiceA.text);

PlayerPrefs.SetString("Question 5 Option 3 Choice B", q5Option3ChoiceB.text);

PlayerPrefs.SetString("Question 5 Option 3 Choice C", q5Option3ChoiceC.text);

PlayerPrefs.SetString("Question 5 Option 3 Choice D", q5Option3ChoiceD.text);

PlayerPrefs.SetString("Question 5 Option 3 Correct Answer", q5Option3CorrectAnswer.text);

}

}

1. Script for puzzle 1

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Puzzle1 : MonoBehaviour

{

public GameObject resistor10000;

public GameObject resistor5700;

public GameObject resistor400;

public GameObject resistor700;

public GameObject resistor25000;

public GameObject resistor1200;

public GameObject resistor990000;

public GameObject resistor3000;

public GameObject resistor60;

public GameObject resistor4200;

public GameObject[] resistors;

public GameObject puzzle1R1;

public GameObject puzzle1R2;

public GameObject puzzle1R3;

public GameObject puzzle1R4;

public GameObject puzzle1R5;

public GameObject puzzle1R6;

public GameObject puzzle1R7;

public GameObject puzzle1R8;

public GameObject puzzle1R9;

public GameObject puzzle1R10;

public GameObject randomResistorParent;

private GameObject randR1;

private GameObject randR2;

private GameObject randR3;

private GameObject randR4;

private GameObject randR5;

private GameObject randR6;

private GameObject randR7;

private GameObject randR8;

private GameObject randR9;

private GameObject randR10;

private float r1Resistance;

private float r2Resistance;

private float r3Resistance;

private float r4Resistance;

private float r5Resistance;

private float r6Resistance;

private float r7Resistance;

private float r8Resistance;

private float r9Resistance;

private float r10Resistance;

private float totalResistance;

public bool puzzle1Complete = false;

public GameObject puzzle1EnergyShield;

public GameObject q2Key;

void OnEnable()

{

foreach (Transform child in randomResistorParent.transform)

{

Destroy(child.gameObject);

}

resistors[0] = resistor10000;

resistors[1] = resistor5700;

resistors[2] = resistor400;

resistors[3] = resistor700;

resistors[4] = resistor25000;

resistors[5] = resistor1200;

resistors[6] = resistor990000;

resistors[7] = resistor3000;

resistors[8] = resistor60;

resistors[9] = resistor4200;

RandomizeResistors(resistors);

ResistorSpawn();

}

void Update()

{

if (randR1 == null)

{

r1Resistance = 0.0f;

}

if (randR2 == null)

{

r2Resistance = 0.0f;

}

if (randR3 == null)

{

r3Resistance = 0.0f;

}

if (randR4 == null)

{

r4Resistance = 0.0f;

}

if (randR5 == null)

{

r5Resistance = 0.0f;

}

if (randR6 == null)

{

r6Resistance = 0.0f;

}

if (randR7 == null)

{

r7Resistance = 0.0f;

}

if (randR8 == null)

{

r8Resistance = 0.0f;

}

if (randR9 == null)

{

r9Resistance = 0.0f;

}

if (randR10 == null)

{

r10Resistance = 0.0f;

}

totalResistance = r1Resistance + r2Resistance + r3Resistance + r4Resistance + r5Resistance + r6Resistance + r7Resistance + r8Resistance + r9Resistance + r10Resistance;

if (randomResistorParent.transform.childCount == 5)

{

if(totalResistance == 5360)

{

puzzle1Complete = true;

PuzzleComplete();

}

}

}

void RandomizeResistors(GameObject[] resistors)

{

// Knuth shuffle algorithm

for (int t = 0; t < resistors.Length; t++)

{

var tmp = resistors[t];

int r = UnityEngine.Random.Range(t, resistors.Length);

resistors[t] = resistors[r];

resistors[r] = tmp;

}

}

void ResistorSpawn()

{

puzzle1R1 = resistors[0];

randR1 = Instantiate(puzzle1R1, new Vector3(15.582f,3.27f, -4.575f), Quaternion.Euler(0,0,90));

randR1.transform.parent = randomResistorParent.transform;

r1Resistance= Single.Parse(randR1.tag);

puzzle1R2 = resistors[1];

randR2 = Instantiate(puzzle1R2, new Vector3(15.582f, 3.27f, -5.138f), Quaternion.Euler(0, 0, 90));

randR2.transform.parent = randomResistorParent.transform;

r2Resistance = Single.Parse(randR2.tag);

puzzle1R3 = resistors[2];

randR3 = Instantiate(puzzle1R3, new Vector3(15.582f, 3.27f, -5.699f), Quaternion.Euler(0, 0, 90));

randR3.transform.parent = randomResistorParent.transform;

r3Resistance = Single.Parse(randR3.tag);

puzzle1R4 = resistors[3];

randR4 = Instantiate(puzzle1R4, new Vector3(15.582f, 3.27f, -6.262f), Quaternion.Euler(0, 0, 90));

randR4.transform.parent = randomResistorParent.transform;

r4Resistance = Single.Parse(randR4.tag);

puzzle1R5 = resistors[4];

randR5 = Instantiate(puzzle1R5, new Vector3(15.582f, 3.27f, -6.827f), Quaternion.Euler(0, 0, 90));

randR5.transform.parent = randomResistorParent.transform;

r5Resistance = Single.Parse(randR5.tag);

puzzle1R6 = resistors[5];

randR6 = Instantiate(puzzle1R6, new Vector3(15.582f, 3.27f, -7.3888f), Quaternion.Euler(0, 0, 90));

randR6.transform.parent = randomResistorParent.transform;

r6Resistance = Single.Parse(randR6.tag);

puzzle1R7 = resistors[6];

randR7 = Instantiate(puzzle1R7, new Vector3(15.582f, 3.27f, -7.948f), Quaternion.Euler(0, 0, 90));

randR7.transform.parent = randomResistorParent.transform;

r7Resistance = Single.Parse(randR7.tag);

puzzle1R8 = resistors[7];

randR8 = Instantiate(puzzle1R8, new Vector3(15.582f, 3.27f, -8.515f), Quaternion.Euler(0, 0, 90));

randR8.transform.parent = randomResistorParent.transform;

r8Resistance = Single.Parse(randR8.tag);

puzzle1R9 = resistors[8];

randR9 = Instantiate(puzzle1R9, new Vector3(15.582f, 3.27f, -9.071f), Quaternion.Euler(0, 0, 90));

randR9.transform.parent = randomResistorParent.transform;

r9Resistance = Single.Parse(randR9.tag);

puzzle1R10 = resistors[9];

randR10 = Instantiate(puzzle1R10, new Vector3(15.582f, 3.27f, -9.643f), Quaternion.Euler(0, 0, 90));

randR10.transform.parent = randomResistorParent.transform;

r10Resistance = Single.Parse(randR10.tag);

}

void PuzzleComplete()

{

Destroy(puzzle1EnergyShield);

BoxCollider[] q2keycolliders = q2Key.GetComponents<BoxCollider>();

foreach(BoxCollider bc in q2keycolliders)

{

bc.enabled = true;

}

}

1. Script for multiple choice system (Question 1)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/\*This is a script for the multiple choice questions found throughout the game. It is extremely

\* versatile as it allows for easy modification to the specifics to each question in the inspector.

\*Requires no alterations in script for each question. It is attached to the desired Screen parent object.\*/

public class Q1MultipleChoiceSystem : MonoBehaviour

{ /\* Text appearing on the cover screen that

appears before the question is started.

Will be used to show the question number.\*/

public Text questionNumberText;

/\* Text object in game that

will display the chosen question string.\*/

public Text questionText;

/\* Child text objects for the

onscreen buttons for the question.\*/

public Text answerA;

public Text answerB;

public Text answerC;

public Text answerD;

/\*Text object that will

display the questionTimer integer.\*/

public Text questionTimer;

// Text object on the correct/incorrect screen that appears after the question is complete.

public Text correctIncorrectText;

public string questionNumber;

// Each multiple choice question has 3 randomized options that can appear when the question is initiated.

// Answer choices for question option 1

private string questionOption1;

private string questionOption1A;

private string questionOption1B;

private string questionOption1C;

private string questionOption1D;

// Answer choices for question option 2

private string questionOption2;

private string questionOption2A;

private string questionOption2B;

private string questionOption2C;

private string questionOption2D;

// Answer choices for question option 3

private string questionOption3;

private string questionOption3A;

private string questionOption3B;

private string questionOption3C;

private string questionOption3D;

// Correct answer(A, B, C, or D) can be chosen in the inspector for each question option.

private string option1CorrectAnswer;

private string option2CorrectAnswer;

private string option3CorrectAnswer;

private string correctAnswer;

private bool correct = false;

private bool incorrect = false;

private bool questionComplete = false;

// Time limit for the player to input an answer

public float questionTimeLimit;

// Bool controlled by a script on the key slot using a trigger collider to detect if the key is inserted by checking the tag of the object inserted.

public bool keyInserted = false;

// Game Objects will be activated when the key is inserted for activation, cover will disappear when "Start Question" is pressed, Correct/Incorrect appear after question is complete

public GameObject questionCanvas;

public GameObject questionCover;

public GameObject correctIncorrectScreen;

//Empty object that holds general scripts. Used to access the health script.

public GameObject generalScripts;

public GameObject rewardedItem;

//Everything inside this function is established when the game starts, including the selected random question and its

//provided answer choices, correct answers, and screen settings.

void Start()

{

questionOption1 = PlayerPrefs.GetString("Question 1 Option 1 Text");

questionOption2 = PlayerPrefs.GetString("Question 1 Option 2 Text");

questionOption3 = PlayerPrefs.GetString("Question 1 Option 3 Text");

questionCanvas.SetActive(false);

correctIncorrectScreen.SetActive(false);

var randomquestion = new string[3] {questionOption2, questionOption2, questionOption3 };

questionText.text = randomquestion[Random.Range(0, randomquestion.Length)];

questionOption1A = PlayerPrefs.GetString("Question 1 Option 1 Choice A");

questionOption1B = PlayerPrefs.GetString("Question 1 Option 1 Choice B");

questionOption1C = PlayerPrefs.GetString("Question 1 Option 1 Choice C");

questionOption1D = PlayerPrefs.GetString("Question 1 Option 1 Choice D");

option1CorrectAnswer = PlayerPrefs.GetString("Question 1 Option 1 Correct Answer");

questionOption2A = PlayerPrefs.GetString("Question 1 Option 2 Choice A");

questionOption2B = PlayerPrefs.GetString("Question 1 Option 2 Choice B");

questionOption2C = PlayerPrefs.GetString("Question 1 Option 2 Choice C");

questionOption2D = PlayerPrefs.GetString("Question 1 Option 2 Choice D");

option2CorrectAnswer = PlayerPrefs.GetString("Question 1 Option 2 Correct Answer");

questionOption3A = PlayerPrefs.GetString("Question 1 Option 3 Choice A");

questionOption3B = PlayerPrefs.GetString("Question 1 Option 3 Choice B");

questionOption3C = PlayerPrefs.GetString("Question 1 Option 3 Choice C");

questionOption3D = PlayerPrefs.GetString("Question 1 Option 3 Choice D");

option3CorrectAnswer = PlayerPrefs.GetString("Question 1 Option 3 Correct Answer");

if (questionText.text == questionOption1)

{

answerA.text ="A. " + questionOption1A;

answerB.text ="B. " + questionOption1B;

answerC.text ="C. " + questionOption1C;

answerD.text ="D. " + questionOption1D;

correctAnswer = option1CorrectAnswer;

}

if (questionText.text == questionOption2)

{

answerA.text = "A. " + questionOption2A;

answerB.text = "B. " + questionOption2B;

answerC.text = "C. " + questionOption2C;

answerD.text = "D. " + questionOption2D;

correctAnswer = option2CorrectAnswer;

}

if (questionText.text == questionOption3)

{

answerA.text = "A. " + questionOption3A;

answerB.text = "B. " + questionOption3B;

answerC.text = "C. " + questionOption3C;

answerD.text = "D. " + questionOption3D;

correctAnswer = option2CorrectAnswer;

}

questionNumberText.text = "Question " + questionNumber;

rewardedItem.SetActive(false);

}

/\* Use the Update function to wait for the key to be inserted

\* and then initialize the question, running the necessary functions\*/

void Update()

{

if(keyInserted == true)

{

questionCanvas.SetActive(true);

if(this.gameObject.tag == "Start Question")

{

questionCover.SetActive(false);

QuestionTimer();

}

}

}

//Controls the timer at which the player has to submit an answer

void QuestionTimer()

{

if (questionComplete == false)

{

questionTimeLimit -= Time.deltaTime;

int roundedtimelimit = Mathf.RoundToInt(questionTimeLimit);

questionTimer.text = roundedtimelimit.ToString();

}

if(questionComplete == true)

{

questionTimeLimit = 0;

}

if(questionTimeLimit < 0)

{

AnswerCheck();

}

}

/\*Answers are input by pressing one of the 4 buttons on

the tables in front of the question screens. Pressing a button changes

the tag of the canvas item to the corresponding letter.

This function will check the tag of the canvas object and see if it matches\*/

void AnswerCheck()

{

bool hitOnce = false;

correctIncorrectScreen.SetActive(true);

questionComplete = true;

if (questionCanvas.gameObject.tag == correctAnswer)

{

correctIncorrectText.text = "Correct";

correct = true;

/\*This if statement is used so health is subtracted or added once.

Without it, it would add/subtract to the health once per frame since

this function stems from Update function\*/

if (hitOnce == false)

{

Reward();

HealthControl();

hitOnce = true;

}

}

else

{

correctIncorrectText.text = "Incorrect";

incorrect = true;

if (hitOnce == false)

{

HealthControl();

hitOnce = true;

}

}

}

//Getting the question right provides a health increase, while getting it wrong hits the player for 20 health

void HealthControl()

{

if (correct == true)

{

var correctbonus = generalScripts.GetComponent<Health>().playerHealth += 10.0f;

var healed = generalScripts.GetComponent<Health>().healed = true;

}

if (incorrect == true)

{

var incorrectpenalty = generalScripts.GetComponent<Health>().playerHealth -= 20.0f;

var hurt = generalScripts.GetComponent<Health>().hurt = true;

}

}

void Reward()

{

rewardedItem.SetActive(true);

}

1. Script controlling the shooting enemies

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class RangedEnemy : MonoBehaviour

{

public GameObject Player;

private Animator rangedAnimator;

public ParticleSystem laserShot;

public AudioClip laserSound;

public float enemyPlayerDistance;

public float raycastHeight;

public bool inSight = false;

public bool inRange = false;

public bool shotReady = true;

public float shotTimer;

public float shotCooldown;

private float randomDirection;

public bool hitObject = false;

public float health;

// Start is called before the first frame update

void Start()

{

Player = GameObject.Find("BodyCollider");

rangedAnimator = this.gameObject.GetComponent<Animator>();

shotCooldown = shotTimer;

}

private void OnCollisionEnter(Collision collision)

{

if (collision.gameObject.tag == "Pistol Bullet")

{

health -= 5.0f;

}

}

// Update is called once per frame

void Update()

{

enemyPlayerDistance = Vector3.Distance(Player.transform.position, this.gameObject.transform.position);

if (health > 0.0f)

{

if (Player != null)

{

transform.LookAt(new Vector3(Player.transform.position.x, transform.position.y, Player.transform.position.z));

}

else

{

transform.LookAt(new Vector3(transform.position.x, transform.position.y, transform.position.z));

rangedAnimator.SetBool("Dead", true);

}

//Check if player is in sight//

RaycastHit hit;

if (Physics.Raycast(new Vector3(transform.position.x, raycastHeight, transform.position.z), transform.TransformDirection(Vector3.forward), out hit, Mathf.Infinity))

{

if (hit.transform == Player.transform)

{

inSight = true;

}

else

{

inSight = false;

}

}

Debug.DrawRay(new Vector3(transform.position.x, raycastHeight, transform.position.z), transform.TransformDirection(Vector3.forward) \* 10, Color.green);

//Check range between enemy and player//

if (enemyPlayerDistance < 9.0f)

{

inRange = true;

}

else

{

inRange = false;

}

//Check if contacting other objects, in sight, in range

if (hitObject == false)

{

if (inSight == true)

{

if (inRange == false)

{

ApproachPlayer();

}

else

{

PlayerInSightInRange();

}

}

else

{

FindPlayer();

}

}

else

{

AvoidObjects();

}

}

else

{

transform.LookAt(new Vector3(transform.position.x, transform.position.y, transform.position.z));

laserShot.Stop();

rangedAnimator.SetBool("Aiming", false);

rangedAnimator.SetBool("Dead", true);

Destroy(this.gameObject, 5.0f);

}

}

void AvoidObjects()

{

rangedAnimator.SetBool("Aiming", true);

rangedAnimator.SetFloat("X", -1.0f);

rangedAnimator.SetFloat("Y", 1.0f);

}

void FindPlayer()

{

rangedAnimator.SetBool("Aiming", true);

rangedAnimator.SetFloat("X",1.0f);

}

void ApproachPlayer()

{

rangedAnimator.SetBool("Aiming", false);

rangedAnimator.SetFloat("Speed", 0.5f);

}

void PlayerInSightInRange()

{

rangedAnimator.SetBool("Aiming", true);

rangedAnimator.SetFloat("Speed",0f);

rangedAnimator.SetFloat("X", 0f);

shotCooldown -= Time.deltaTime;

if(shotReady == true)

{

if(shotCooldown<0)

{

EnemyFire();

}

}

}

void EnemyFire()

{

laserShot.Play();

AudioSource.PlayClipAtPoint(laserSound, this.gameObject.transform.position);

rangedAnimator.SetTrigger("Shoot");

shotCooldown = shotTimer;

}

}

REFERENCES

[1] Teed, R. (2018, May 07). Game-Based Learning. Retrieved December 12, 2018, from https://serc.carleton.edu/introgeo/games/index.htm

[2] Haberstroh, T. (2017, February 8). Game-Based Learning. Retrieved April 10, 2019, from http://www.espn.com/nba/story/\_/page/presents5/washington-wizards-virtual-reality