Method (what you are going to do)

For this project, I used the Waterfall methodology. The Waterfall methodology has six main steps, requirement gathering and documentation, design, implementation, testing, deployment, and maintenance. (1) In requirement gathering and documentation, we gather comprehensive information about what the project requires and create a requirement document. (1). In design, using the establish requirements, we establish detail specification such as programming language/hardware requirements and specification for the actual implementation. (1). In Implementation, programmers create functional program according to the specifications. (1). In testing, we test the created product and find any bugs or any other serious issues. (1) In delivery, we submit the product to be deployed or released. (1) Finally in maintenance, we make updates and patches based on bugs found or on feedback from the customer/client (1). One of the biggest reasons, I decided to use the Waterfall methodology is because of its emphasis on the requirement gathering and documentation. Since I have little experience making video games, I would need significant research and practice to get a good idea of the scope and requirements.

-explain more about the waterfall method and its history

-Explain what you did each step

1. <https://www.lucidchart.com/blog/waterfall-project-management-methodology>

Game World

Method

-Third-person vs 1st person

-Potentially put the “practice” I did in order to create the project

Result

-maybe explain how unity works like how Hero Training -Unity 2D RPG guy did in the beginning of Result. Start() and Update().

-Parents/child object

-component

-Scene

**Game World**

In Unity

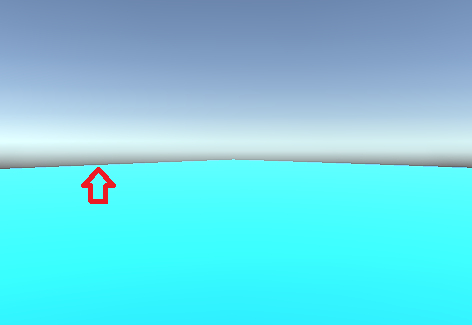
A scene is where the contents of the game, the assets, are stored in Unity. In my project the scene was named “main” and it contains the assets created.

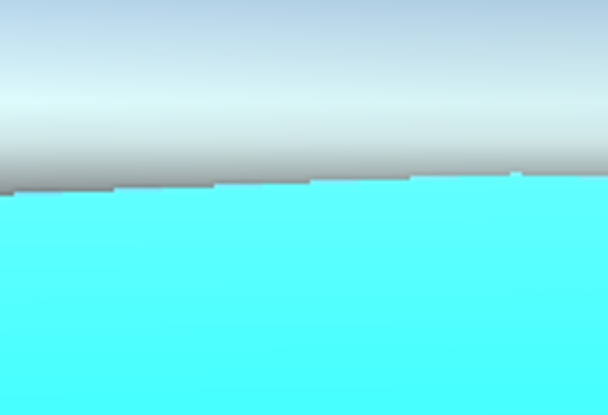
In Unity, in the Lighting panel, the lighting and sky-box can be adjusted. I used the Default-Skybox with Directional Light. The skybox made the game look expansive and the directional light created shadows, which made the game look more realistic. In order to make the color more vibrant, I used “Auto Generate” under “Lightmapping Setting”, instead of manually setting the lighting setting.

The game world is a large square plane. The player’s point of view (the main camera) is set to the center. The color is set light blue using “Materials”. In unity, materials are scripts used to define the looks, such as texture of the object. Light blue was chosen since it gave the game a futuristic look and was relatively easier on the eye compared to other neon colors. Reddish neon colors were not considered since the virus is red and the virus could blend-in with the ground making it more difficult to see from the player.

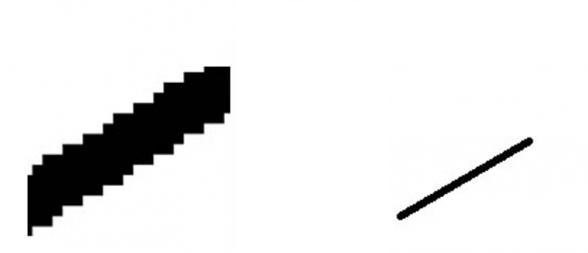
In Unity, “Main Camera” object is used to decide what is shown when the game is run. The camera was set above the center of the plane, facing one side of the plane.

Originally, the plane was scaled by 5. The plane was made small as possible to reduce the size of the game and make the game run smoother by making the game render less. The plane could be small since the player is immobile in the center. However, from the player’s point of view, the edge of the square looked jagged.

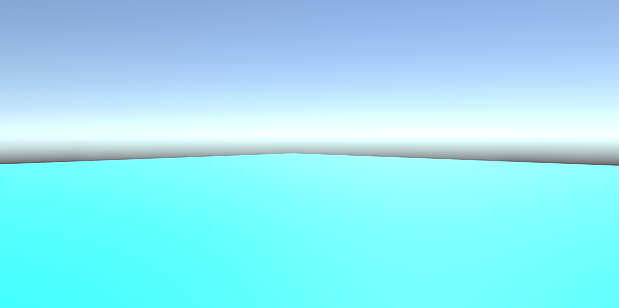




The edges are jagged because images on screen are created using tiny squares called pixels. For example, seen bellow, the close-up of the diagonal line (right) is on the left.

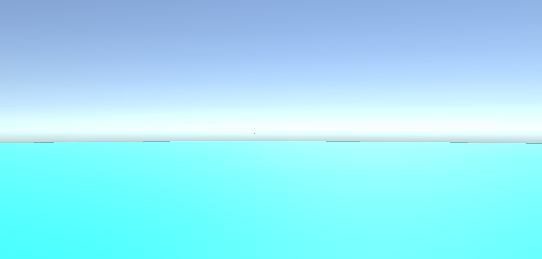


Anti-aliasing was applied in an attempt to make the edges look more straight. Anti-aliasing allows the edge of each pixels to blend in with one another, blurring them. This technique creates a smoother-looking edges. In the project setting under “Quality”, we can change the degree of anti-aliasing. There are four degrees, Disabled, 2x Multi Sampling (the default setting), 4x Multi Sampling, and 8x Multi Sampling. The 8x Multi Sampling did improve the look of the edges.



However, we could still see the aliasing and anti-aliasing requires more resource to render objects, I decided to set back the anti-aliasing to the default, 2x Multi Sampling.

A spherical plane was also tested. Since unity does not provide a spherical plane, a sphere was flattened by scaling the y-axis by 0.01 and x/z-axis by a 100. But black line appeared on the edges. Scaling the sphere-plane by 200 produced similar aliasing issue.



Instead, the plane was scaled by 20, eliminating the jagged edges. The plane is large enough that from the player’s point of view, the plane seems to go beyond the horizon. Since the player cannot move, this large square suffices as the game world.

-Aliased vs anti-aliased image

<https://pclab.pl/art75403-2.html>

**Player**

A capsule was used to represent the player. Capsule seemed the most appropriate as it had the most humanoid-shape compared to other 3d objects, such as sphere, square, and cube, provided by Unity. There is a “ragdoll” 3d object option that allows users to specify detailed parts of a body, such as the legs, torso, head, and total mass etc. But since the player will be immobile in the center, it was not considered. An empty object Player was created and made the parent object of Capsule for organization.

**Camera Movement**

The camera moves according to the mouse movement. If the mouse moves right, the camera turns right and if it moves up the camera turns up etc.

For the camera movement, I used code that I found from a youtube video. <https://www.youtube.com/watch?v=lYIRm4QEqro>

A C# script, Camera.cs was created and was attached to the Main Camera.

The mouse can move in two directions, horizontally (along the x-axis) or vertically (along the y-axis). In order to turn the camera horizontally, the camera is rotated on the y-axis according to the mouse’s horizontal movement, according the x-coordinate. In order to turn the camera vertically (up and down), the camera is rotated on the x-axis according to the y-coordinate of the mouse. Transform.eulerAngles represents three-dimensional rotation around the z, x, and y-axis in that order.

When running the game on unity, we could not turn the camera when the mouse cursor moved outside of the “Game” window, where you run the game. Furthermore, the mouse cursor was visible. This was a huge annoyance as the mouse cursor would need to be dragged to the center or inside the “Game” window continuously. An empty object Game Manager was added to the game world. GameManager.cs was attached to it. Cursor.lockState = CursorLockMode.Locked; was added to the Start() so that the cursor icon disappeared and the cursor was locked in the center, when the game is run.

**Bullet**

The bullet asset was created using a Prefab. In Unity, prehabs are reusable assets that allows for creating and storing game object with all the components and property values. This prevents making the same edit to every copy as they are all kept in sync when using Prefabs. I created a cube with red material and added a Trail Renderer component to add a trail to make the bullet more interesting than just a cube.

A c# script Bullet.cs was created and attached to the prehab. In Start() the velocity is set with

rb.velocity = transform.forward \* speed;

Transform.forward returns a normalized vector representing the z-axis. This direction is multiplied by the speed to set the velocity of the Rigidbody of the bullet, which moves the bullet in a forward direction. A Rigidbody component applies Unity’s physics engine to the object attached and also allows the user to tune its affect on the object. While the velocity could be set in Update(), I called it in Start() as the bullet only goes in one direction and setting the velocity every update for multiple bullets use unnecessary resource.

In order to shoot the bullet with mouse click, a long cuboid with green material was created to represent a gun and was positioned around the middle of the capsule. An empty object fire position was also created to set the position in which the bullet will be fired. The gun and the fire position was made to be the child of Main Camera so that the gun will move as the player moved the mouse to look around.

Player.cs was added to the Player. The Update() function will constantly check for a mouse click with if (Input.GetMousebuttonDown(0)). Once the mouse is clicked, Instantiate is called to create a bullet object from the Fire Position that we created before.

One of the problems encountered was how the bullet was never destroyed once they were shot. They flew outwards indefinitely until the game was ended. A countdown mechanics was implemented to destroy the bullet after a set time.

Games are presented on screen according to the frame rate, the frequency at which images (frames) are displayed on screen. In Unity, the Update() function is called at every frame. However, frame rate is affected by the specs of the computer and the refresh of the display, so it varies according to the user’s machine. Say in Update() an object was moved by 1cm, a user with 60 frames/sec would move 60 cm every second but a user with 30 frames/sec would move by 30cm. Even on the same machine, framerate fluctuates as it is affected by cpu and gpu usages. To be independent from framerate, Time.deltaTime was used. Time.deltaTimes is the amount of the time the last frame took to complete.

bullet\_duration = bullet\_duration - Time.deltaTime;

Subtracting Time.deltaTime from a duration, would ensure that a countdown will be consistent amongst different machines. When the bullet\_duration reaches zero, the bullet is destroyed by calling the Destory method.

Another problem encountered was how when turning the camera and shooting. The gun and the bullets would flicker when moving. This was because the code that moved the camera in Camera.cs was inside Update(). In Unity, the Update() method is called for all the object in unknown order. The reason for the flickering was because the camera moved slightly earlier than the bullet before the bullet location was updated so the bullet appeared in space. A better approach would be for the bullet to be first updated to the new location and the camera turns to capture the bullet. Instead of using Update(), LateUpdate() was used to in Camera.cs to move the camera. LateUpdate() is called after all Update() functions are called.

**Enemy**

Initially, a sphere 3d-object was used to represent the enemy. RigidBody component was added to apply movement. Virus.cs was attached to this object, where in Start() the virus is faced to the center and the velocity is set to move towards the center. Instead of using more resource by constantly setting the same velocity in Update(), it was set in Start() since the virus is moving in one direction towards the center with the same speed.

When running the game, the sphere jittered as it approached the player. This was fixed by changing the interpolation of the RigidBody from “none” to “Extrapolate”. According to the Unity documentation about Rigidbody.interpolation, physcis is running at discrete timesteps, while graphics is rendered at variable frame rates. “This can lead to jittery looking objects, because physics and graphics are not completely in sync.”

<https://docs.unity3d.com/ScriptReference/Rigidbody-interpolation.html>

**Player and Enemy Interaction**

One of the problems encountered was when the virus collided with another virus, they disappeared. The objects were destroyed upon contact. Three different solutions were considered. The first was turning the virus’s Sphere Collider off. Colliders are used to define the area of an object where if another object’s collider intersected, a collision event is created. However, since the virus need to interact with other objects such as bullets and player, this options was not optimal.

Another option was to create a layer for the virus and change the physics setting so that the object in the virus layer will not interact with other viruses since they are in the same layer. In Unity, we can create layers and assign them to objects. For example, the ground plane and rocks can be put in a “World” layer, the enemies could be put in the “Enemy” layer etc. Virus layer was created and in Edit -> Project Settings -> Physics the layer collision matrix can be changed to set whether each layer collides with the other layers or itself. Once the duplicated virus objects were assigned to the “Virus” layer, in the layer collision matrix, the setting was changed so that the “Virus” layer did not collide with itself.

Ultimately, I made the virus object into a trigger. In a Collider component, we can check “Is Trigger”. Triggers are ignored by the physics engine, so an object would not physically collide with another object.

**Score System**

Whenever the bullets collides with the virus and destroys it, the player will receive a point. In order to do so, an int variable, current\_score was created in Player.cs. AddPoints(int points) adds the points to the current\_score.

The bullet object needed to be able to update the current\_score and the game over scene that will be shown after the game needs access to the current\_score since it will show the player’s final score. Since two objects needed to have access to Player.cs and there is only a need for one instance of Player.cs since this is a one-player game, instead of assigning instances of Player.cs to those two objects, Player.cs was made into a static instance.

public static Player instance;

The above line was added a variable in Player.cs. “Static” indicates that every instance of a class will contain the exact same pointer to a variable.

private void Awake()

{

instance = this;

}

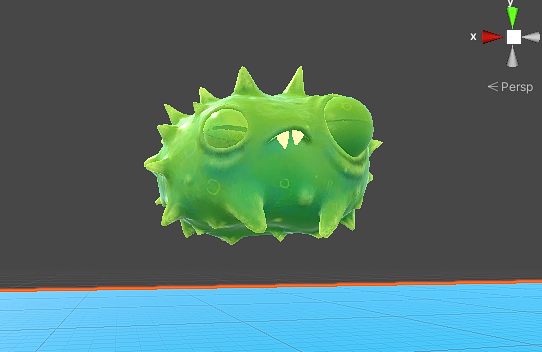
Awake() function, as seen above, was added to Player.cs. What Awake() functions do is that they are called before Start() functions. “this” is the instance of the Player.cs. The specific instance of the Player.cs when the game is run is set to the static variable “instance”. This allows other classes to access the functions and variables of Player.cs.

**Virus and Gun Models**

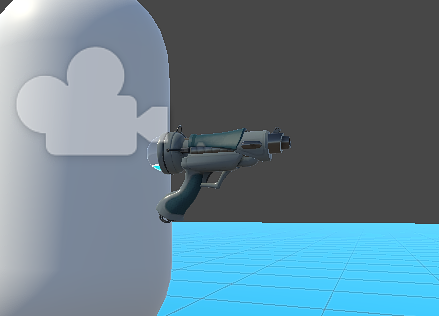
Initially, instead of using detailed 3D models, a sphere was used to represent the virus and an elongated-cube was used to represent the gun. This was done since I was inexperience with game developing, so I did not know how long it will take to implement basic mechanics such as shooting and camera movement. Working with simple shapes made it easier to create the basics of the game. Furthermore, I had not experience creating or working with 3d-models. Thus, the plan was to add detailed 3D models after the basics of the game were implemented and there was still time left.

A 3D model that looked like a virus was needed. Creating from scratch using Blender, a free and open-source 3D modeling software, was considered but the UI, audio, exporting, testing, and writing the report needed to be done, so learning another software from scratch, creating the asset, and then incorporating it to the game seemed too time-consuming. Thus, I looked for a free virus-model online but could not find any. There were a few models on Unity Asset Store, but they were not free. Ultimately, I used a 3D model in “Creator Kit: FPS” on Unity Learn, a tutorial platform that Unity offers.

The asset for the creator kit was downloaded using Unity’s package manager. The Germ Spike prehab and the gun, Healmatic500 were used. The Germ Spike has an animation where it bobs up and down and makes a continuous bumbling sound audible from a close-enough distance.



Germ Spike prefab



Healmatic500

A sphere collider, rigidbody, and the virus.cs script were added to the Germ Spike prehab. The Germ Spike prehab was tagged as “Virus”. The center point that the virus aim towards were lowered since the new virus with 3D model seemed to approach higher in the player’s view. In other words, at

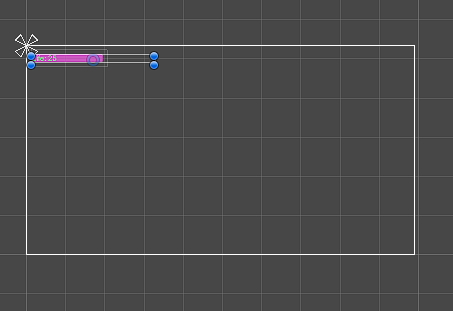
transform.LookAt(new Vector3(0.0f, 1.6f, 0f));

in Start() function of Virus.cs, 1.6f was changed to 0.5f.

-if you want you can add how you had difficulty importing, since I tried to not download unneeded assets, but I just decided to download the whole thing.

**UI**

In order to display the amount of life that the player has left, a UI that shows how much life the player has was created on the top left corner. In Unity, a “Canvas” is an rectangular area that includes all UI element. The first child element of the canvas in the hierarchy is drawn first, then the second child is drawn next and so on. A slider was used to show how much life the player has left and another text component was drawn on top by making the text component the second child of canvas after the slider.



A canvas



Canvas in the hierarchy

In order to set the value of the slider, two new UI variables health\_bar of class Slider and life\_text of class Text were added to Player.cs. The Slider and Text class are included in a native library called UnityEngine.UI, which was imported in the beginning. The Slider and Text component from the hierarchy were assigned to those variables. In Start() the life\_text and health\_bar slider’s values are set. In order to update the value whenever the player is hit by the virus, in the TakeDamage() function those two components are updated.

Another text-UI component was added to show the player the score and was placed in the top center of the screen. The score is updated in AddPoints() function in Player.cs, showing the current\_score.

**Menu Screen**

In order to create a start menu, a new Scene called menu was created. In the new scene, a Canvas component contains a text component that shows the game title and two button components, which starts the game and quits the game.

A new script menu.cs was created. This script contains two public functions StartGame() and QuitGame(). StartGame() using the native SceneManager class’s LoadScene() function to load the “main” scene, which contains that main game. Initially, call the LoadScene() method returned an error but that is because I was missing a crucial. In order to load scenes, we need to go to File->Build Settings and add all the scenes used in the game.

The QuitGame() function calls the Quit() function in the native Application class to shut down the game.

**Pause Screen**

The pause screen was created so that the player can stop playing for a moment, quit the game, or restart the game. The pause screen is activated by pressing the escape key in game. This action is registered in the Update() section in GameManager.cs if the pause screen is not already active. The pause screen was made in a similar fashion to the menu screen, but instead of making a new scene it was made inside the canvas of the “main” game scene. A new public GameObject variable pause\_screen that contains the pause screen component and Pause(), Unpause(), MainMenu(), Restart(), and QuitGame() were added to the GameManager.cs. The Pause() function activates and makes the pause screen visible and unlocks the cursor so it can freely move. Then sets Time.timeScale to 0f, which basically freezes the since Time.timeScale controls how fast objects move in the game. The Unpause() function deactivates the pause screen, locks the cursor to the center and sets Time.timeScale back to 1f. However, since the shooting mechanic and camera movement is based on mouse movement and not on time, the player can move the camera and shoot a stationary bullet that doesn’t move in pause screen. So a guards were set so that the player can shoot or move the camera only if the pause screen is not active.

The resume button was made larger and placed in the center than the other buttons so that the player does not accidently click other options and lose the progress of the game.

**Game Over Screen**

The game over screen shows the score of the user.

**-Can potentially add about how you can make other menu scripts use menu.cs**

**Audio**

-can talk about the master mixer etc and how the audio section works

**BGM**

In order to keep a player engaged to the game, the player needs to be stimulated in different ways as possible. Background music adds another layer of stimulation. I looked for different free background music. There were many great songs on Sound Cloud, a prominent online audio distribution platform. Songs that are too complicated or lyrics were avoided since it could distract the player from the game and could potentially mask the sound effects in game. BGM needs to be simple enough that different kinds of players will not have any problem, but needed to be engaging enough to entice them.

Ultimately, 8-bit music were used as the BGM. 8-bit music, also known as “chiptunes”, are a genre of electronic and synth-music that sounds as though they were made using computer chips. The 8-bit music fit the futuristic color scheme that was used in game. Parts of 8-bit-styled-song tend to have simple and repetitive musical notes so it did not distract the player. There were many great 8-bit BGM so I decided to use different songs for each scenes, main menu, the actual game, and the game over scene. For the main menu scene, “Underclocked” by Eric Skiff was used since the song conjured a feeling of anticipation, which fit the anticipation that player would feel when the player is about the start the game. During the game, “Powerup!” by Jeremy Blake was used since it has a relatively upbeat tone. An upbeat-tone is better than a sadder-tone since the player will be listening to it repeatedly and the player will less likely to be frustrated. For the game over scene “A night of Dizzy Spells” by Eric Skiff was used since it has a more toned-down, sadder feeling, which was suiting since the player failed.

**Sound Effect**

**Menu Sound Effect**

Sound effects were added when clicking options on menu. When the player presses, “Restart” on the pause and game over menu or “Start” button on the main menu, a short and upbeat sound effect plays. When pressing “Quit Game” button or the “Main Menu” button on the pause menu, a more muted, in a way, sadder tone plays. I used to two different sound effect to add variation and in the hopes that a more upbeat sound effect may slightly encourage players to play again and a sadder tone may slightly discourage players from quitting. The sound effect were found on freesound.org, a website with CC licensed audio samples.

In order to finish playing the sound effect before switching/loading scenes and quitting the game, the game needs to wait for a specific time. Coroutine was used. “A coroutine is a function that allows pausing its execution and resuming from the same point after a condition is met.”(1)

1: [**https://www.javatpoint.com/unity-coroutines**](https://www.javatpoint.com/unity-coroutines)

The sound effect is played and the WaitForSeconds() function is called to wait for a given amount of seconds. After those seconds, a scene is loaded or the game is quit.

WaitForSeconds() worked for the main menu and the game over scene but did not work for the pause menu. The function only started waiting after the screen was un-paused. This was because in order to stop the viruses from moving the Pause() function in GameManager.cs scales the time to 0f, stopping the flow of time in the game. WaitForSeconds() stops time based on scaled time, that is why it started to execute after the Time.timeScale was set back to default of 1f. WaitForSecondsRealtime() was used instead of WaitForSeconds() since the former uses unscaled time. The time scale was set to the default of one in the start of the main game, since the time scale was set to 0f in at the pause menu.

-can possibly elaborate on how coroutines was implemented

-can possible write about how I created SFXController.cs

**Game Sound Effect**

The sound effect for shooting, player taking damage, and virus exploding were implemented. The audio sources were found on freesound.org. The shooting sound effect sounds like a laser beam to fit with the futuristic theme. Instead of using an actual explosion sound, I used the sound of breaking glass. I wanted a crisp and lighter sound since the player will be constantly listening to it and the heavy sound of an explosion may not be comfortable.

**Audio Management**

Because of the growing number of the audio files used for the game and different scenes sharing the same audio files, I decided to make a static class that manages all the audio files for better code design. Instead of assigning the audio files directly to different files, each scene will only need an empty object with the AudioController.cs and can play the sound from any other scrips.

**Explosion Effect**

The explosion effect was one of the free effects in Shrebb’s Particle Collection, which was found on the unity asset store. The particle effect could be created in Unity but having access to a collection of effects made it fast and easy to compare and contrast different kinds of explosion, especially since I did not know what kind of explosion I wanted. I chose Shrebb’s package since it contained sci-fi themed explosion which matched the theme of the game.

**Export**

For the initial export, the menu screen was too small and the cursor was visible in the center during the game. The visibility of the cursor was set to false at the start of the game scene to hide the cursor by adding “Cursor.visible = true;” in Start() of GameOver.cs.



The Menu Screen During the Initial Export

In the canvas object, where the UI is located, the Canvas Scaler component sets the scale of the UI. For the “UI Scale Mode” section I choose “Scale With Screen Size” option instead of Constant Pixel Size. This made it so that the UI stretches according to the user’s screen size instead of staying at a constant size. The Reference Resolution, the resolution the UI layout is designed for, was set to 800 by 600. This valued was decided though testing out different aspect ratio and screen size and 800 by 600 looked the best.

The game is exported as a full screen game since the windowed border makes it seem less immersive to the player.

**Testing**

We tested phase 1 with my father and a friend of mine. My father said its hard to know where they are coming from and a radar-like system would help. A friend who has a not so optimized computer said that the lap has a difficult time running the game.

**Phase 2**

Now the basic game was done we got to phase 2 with new additional game components.

Testing

**Cursor**

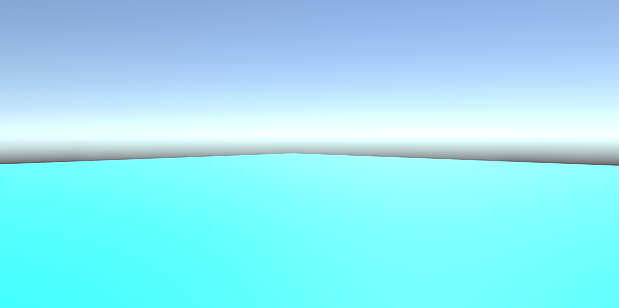
The cursor is located in the center of the screen. In order to shoot the bullet to where the cursor is pointing ray cast was used. Ray cast is an imaginary line from a position to another. If the ray cast intersect with an object, the bullet will shoot to that position. In Player.cs, a Ray object was created with

ray = main\_cam.ScreenPointToRay(new Vector3(x, y, 0));

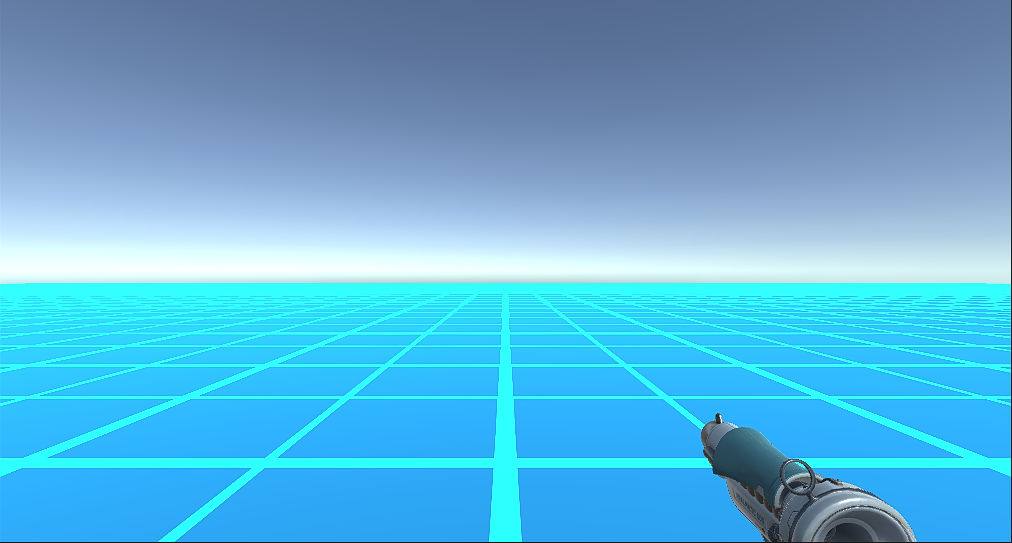
The Unity native function ScreenPointToRay in Camera class returns a ray that starts from a specific position on screen, in our case, the center of the screen. The Physics.RayCast(ray, out hit, distance) function returns a boolean if the “ray” parameter hits an object within the “distance” and outputs the information to “hit”, RaycastHit object. If the ray does not hit anything, the bullet is shot forward to the position into the distance.

**Skybox and Plane Material**

The aesthetic of the game world was bland since phase one was focused on the basic implementation of the game. To give a more futuristic look to the game, a material with tile like feature found in NeonSphere – Starter Pack, a free asset pack from the unity asset store, was added to the plane. The tiles were made smaller to create a grid-like structure and colors were changed to neon blue.

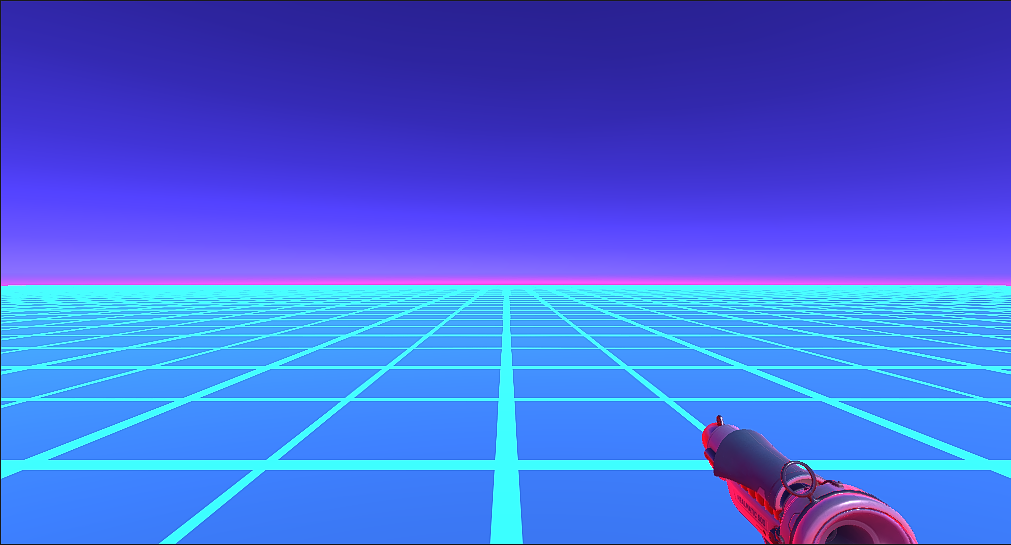


Before the Material



After Tile-Like Material Added

For the skybox instead of using the default sky box, a new neon purple sky box was created. The horizon was made more pinkish to add to the futuristic style.



With The New Neon Skybox

**Menu and UI**

Changed the font and the background scene to the above. I got rid of the UI and the gun to make the picture less cluttered. I also deleted the game over scene and added a UI component to the main game scene, so that the game over ui will show at the moment of players death.