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

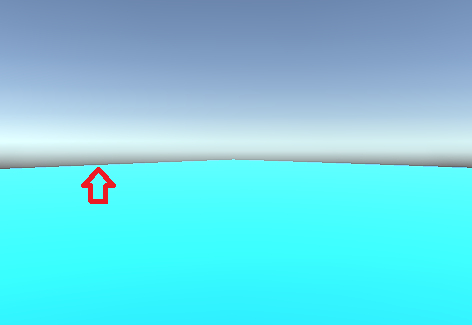
A scene is where the contents of the game, the assets, are stored in Unity. In my project the scene was named “main” and it contains the assets created.

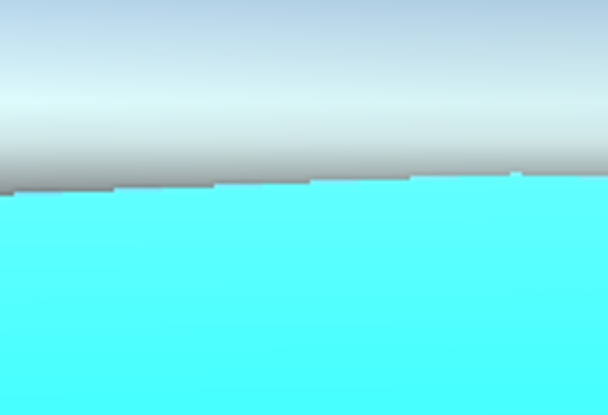
In Unity, in the Lighting panel, the lighting and sky-box can be adjusted. I used the Default-Skybox with Directional Light. The skybox made the game look expansive and the directional light created shadows, which made the game look more realistic. In order to make the color more vibrant, I used “Auto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

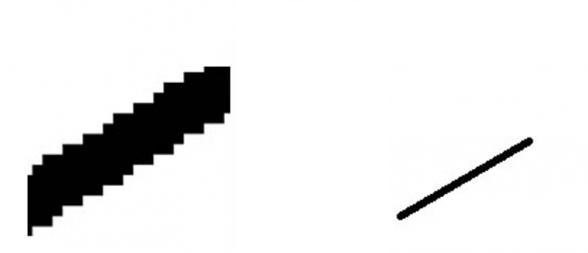
In Unity, “Main Camera” object is used to decide what is shown when the game is run. The camera was set above the center of the plane, facing one side of the plane.

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.

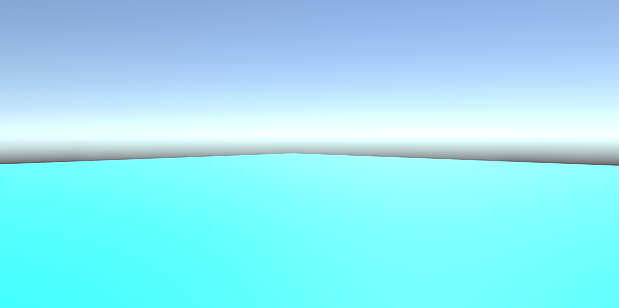




The edges are jagged because images on screen are created using tiny squares called pixels. For example, seen bellow, the close-up of the diagonal line (right) is on the left.

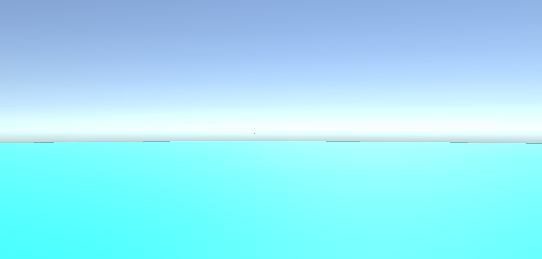


Anti-aliasing was applied in an attempt to make the edges look more straight. Anti-aliasing allows the edge of each pixels to blend in with one another, blurring them. This technique creates a smoother-looking edges. In the project setting under “Quality”, we can change the degree of anti-aliasing. There 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.

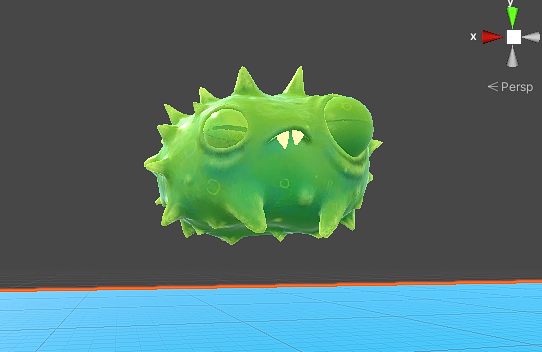
Ultimately, I made the virus object into a trigger. In a Collider component, we can check “Is Trigger”. Triggers are ignored by the physics engine, so an object would not physically collide with another object.

**Virus and Gun Models**

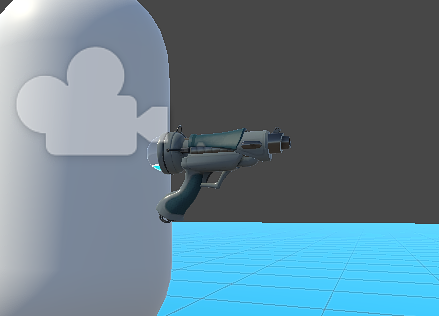
Initially, instead of using detailed 3D models, a sphere was used to represent the virus and an elongated-cube was used to represent the gun. This was done since I was inexperience with game developing, so I did not know how long it will take to implement basic mechanics such as shooting and camera movement. 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

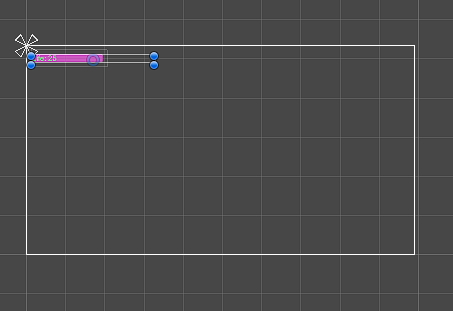
transform.LookAt(new Vector3(0.0f, 1.6f, 0f));

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.

**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.