Focus Track: New Technology Project

Beibei Ji

S434608

Hanze University of Applied Sciences

3rd of February 2023

Contents

Introduction 3
Research 3
Technical Design 5
System Requirements 5
Game Flow 5
Class Diagram 5
UI 6
Low-Fi & High-Fi 6
Low-Fi 6
High-Fi 7
Sound 8
Version Control 9
User Guide 9
Reflection 9
References 10

Introduction

The case to be discussed here will be ‘Radar System’ and during this whole process, I will be using an existing code base FPS Game from Unity Learn (Learn, n.d.). This case was selected because it seemed interesting as I did not create radars before, and I would work with camera which I have great difficulties with. Additionally, I had a chance of creating my own shader for highlighting objects since I always used a free asset “Quick Outline” from Unity Store (Nolet, 2022). Considering the other cases, radar system was not too difficult nor too easy case for me to try to work on.

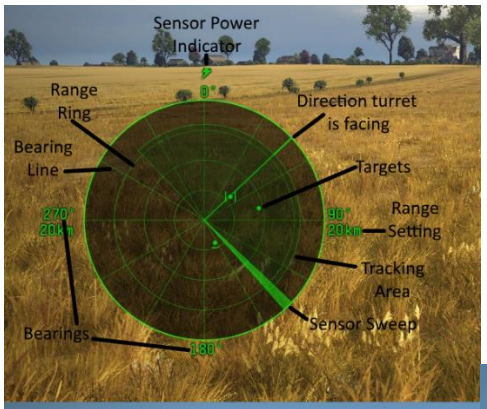
My current skills in programming are a bit below-average in my opinion. The reason for that is that even though I have experience in programming, I am knowledgeable about basic information any programmer should have and am capable of using them in real-life practice. However, creating simple solutions to complex problems are quite impossible for me. I would probably use too many variables, if statements, and loops instead of creating functions for specific events, state machines, etc. Even though I know about them and that they would be useful in the long run, I am not capable to use them voluntarily.

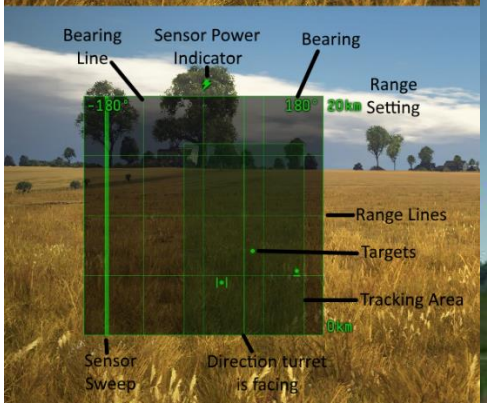
My goal for this Focus Track is to optimize the code, if possible iterate it into an existing code inside the project without doing many changes, and to understand more concepts enough to be confident in using them.

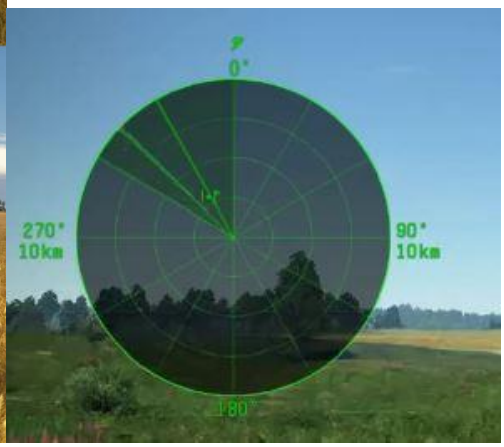
Research

I looked into features of what approximately should a Radar System have. I found a few games with interesting radar systems like War Thunder and Alien: Isolation.

War of Thunder has a few Radar System types to speak of like track radar that scans the area but when it locks to a target, all other targets are removed from the UI. Some can even scan a specific area between two angles in the Radar System for a thorough scan of the area (Wiki, n.d.).







Alien: Isolation, on the other hand, had a motion tracker showing targets only when they moved. Unlike usual, it did not have a radar sweep and had to be picked up by the player using their hands to see the screen (Fandom, n.d.).

Apart from that, I did look at some YouTube tutorials as well as I was not sure about how to code it in case the game didn't have a Radar System. Most of them were only for a reference for what were they steps in creating a Radar System out of scratch. The most useful tutorials were from Code Monkey (Monkey, 2019), and Andrei Taranovskii (Taranovskii, 2020) Other tutorials were dedicated for me to learn how to code Unlit Shaders in Unity. Albeit unsuccessful, I did learn about the structure of Unlit Shaders and its properties thanks to Madalaski (Madalaski, 2020) and Unity Manual (Documentation, n.d.). There was also a very good introduction to drawing an outline by Alexander Ameye (Ameye, n.d.) but I did not fully understand how I was supposed to implement it from those concepts so it would work.

From those, I could have a clearer idea of what features are required for a Radar System to be recognized as one:

- A specific map – either a section or a whole area around the player.
- If targets are detected, the location should be shown relative to the location of the player

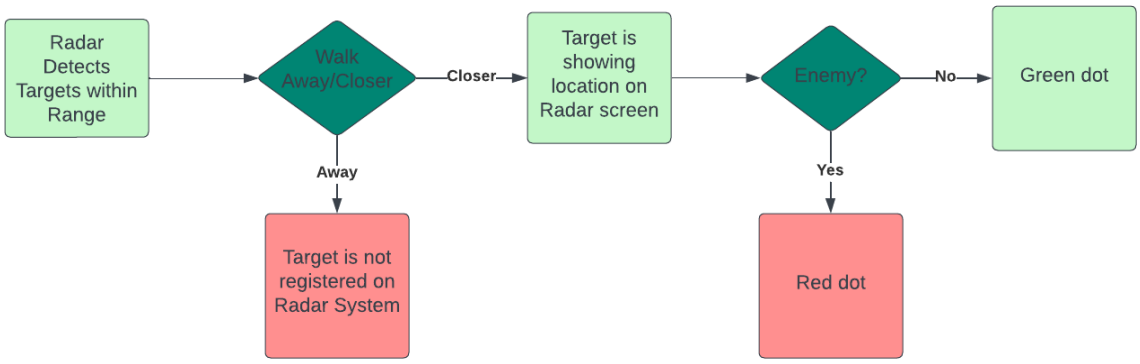
Technical Design

System Requirements

- Radar
  - Register targets
  - Show them on screen
  - Differentiate between targets
- HUD
  - Show differences between targets on the screen
- Easily add objects to radar detection

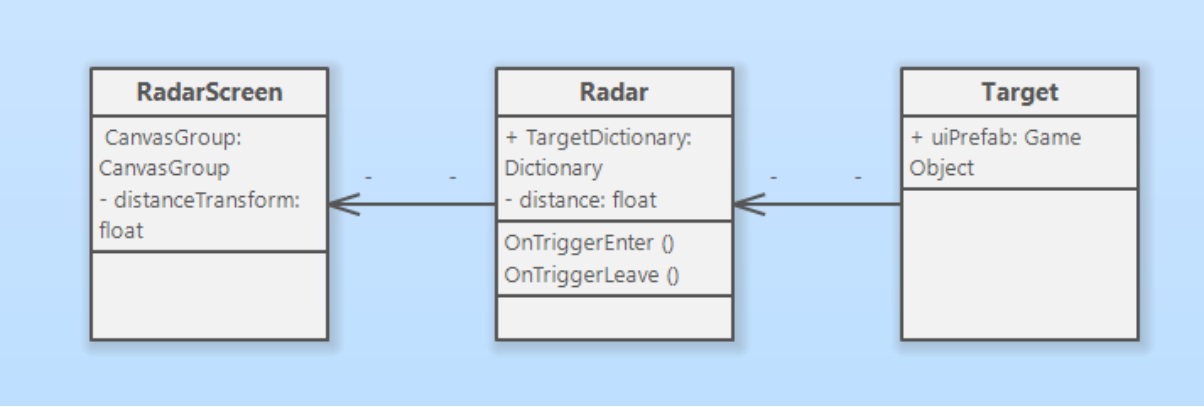
Game Flow

The Radar System is simple. If the specific targets that Radar identifies are within range of the player, their location would be displayed on a Radar screen and highlight the closest target to the player in World Space.

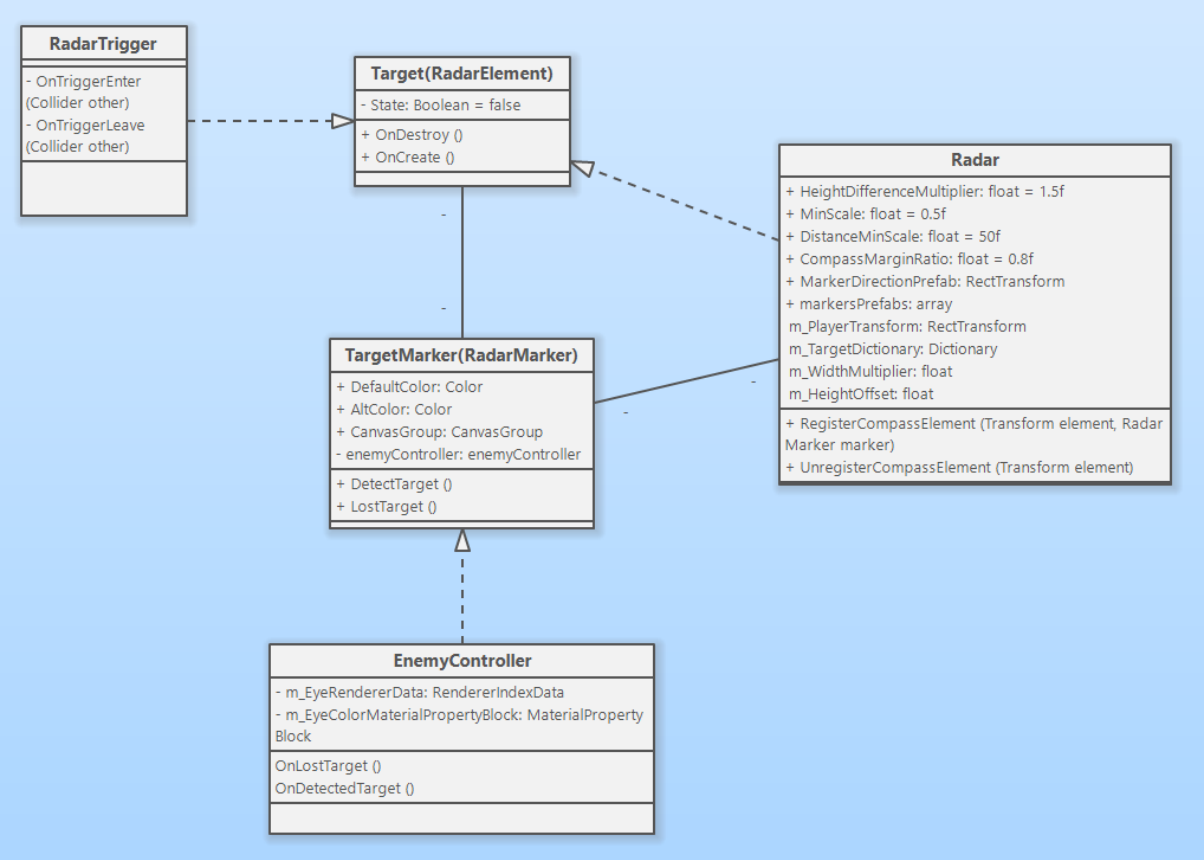


Class Diagram

For a basic visualization of the classes, I first made a very simple Class Diagram before coding:



After coding, however, it did become a bit more complex as I also used a pre-existing code so that it functions as shown below:

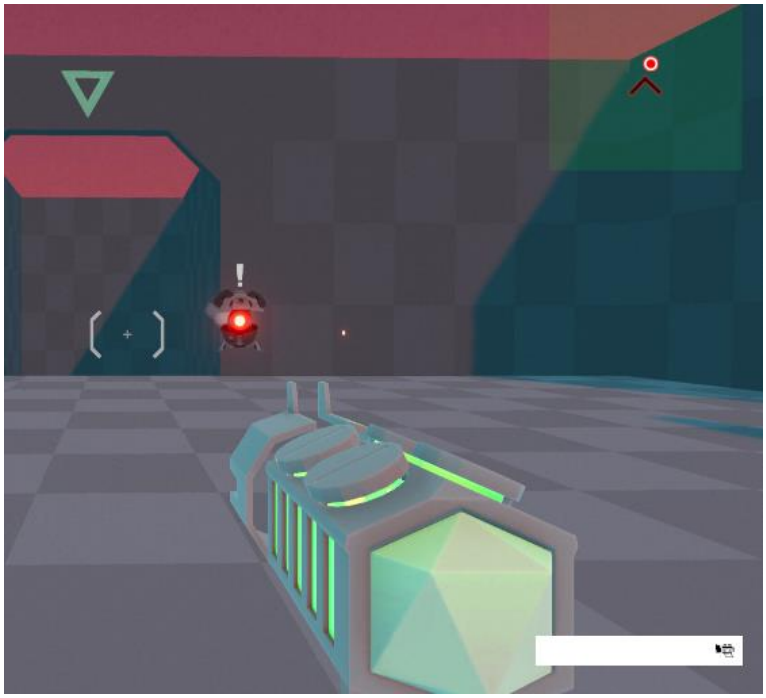


UI

Low-Fi & High-Fi

Low-Fi

I duplicated the scripts and their Assembly Definition Asset from the Compass-related scripts and UI. After a few changes to how the locations of targets are showing to top-view instead of front-view, the Radar System was working well as shown below.



However, after some time, I noticed some flaws. Using a dictionary and the scripts for the compass, the targets were stored in the Radar System from the beginning. It would be easy to add more targets, but I chose to adjust the scripts a bit as I suppose it would not be as efficient when we want to track many objects in the game.

Therefore, I created another script apart from the namespace to create a trigger function to register objects whenever they are within the player's box collider.

```
public class RadarTrigger : MonoBehaviour
{
    // Start is called before the first frame update
    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "target" || other.gameObject.tag ==
"pickups")
        {
            other.GetComponent<RadarElement>().OnCreate();
        }
    }

    private void OnTriggerLeave(Collider other)
    {
        other.GetComponent<RadarElement>().OnDestroy();
    }
}
```

High-Fi

After completing the Radar System, I added an array of markers to differentiate between enemies and other detectable objects. Each object would then be granted a tag and based on the tag would have a specific marker on the Radar screen.

```
        public void OnCreate()
        {
            if(!awoken)
            {
                var i = 0;
                //var markerInstance = Instantiate(CompassMarkerPrefab);
                switch (tag)
                {
                    case "target":
                    i = 0;
                    break;

                    case "pickups":
                    i=1;
                    break;
                }
                var markerInstance = Instantiate(m_Compass.markersPrefabs[i]);
                m_Compass.RegisterCompassElement(transform, markerInstance);
                awoken = true;
            }
        }
```

Along with that, I added a simple sweep animation to make it look more like a Radar System. The screen does not rotate according to the player's rotation but the player's marker does to familiarize the player. It is also possible to add Compass next to it for better orientation. However, further iteration on the compass script is needed as I did not touch it.

Sound

The sound used for the Radar System was the same as the sounds for other UI buttons. I applied sound effects to the Radar sweep over the screen.

Version Control

For version control, I used GitHub Desktop the published repository can be accessed through this link: https://github.com/Kirirgiri/newTech.git

User Guide

The radar is an in-built feature of this first-person shooter and is used as a device to give players a visual representation of their surroundings. The radar works in a 360-degree view around the player and detects enemies once they come within range. There is a list of elements that are displayed on the radar when detected by the sensor:

- Enemies
    - Hoverbots
    - Turrets
- Other detectable targets
    - Heal pickups

Depending on the object, the dot's characteristics on the Radar Screen will vary:

- Enemies: red dot
- Other detectable objects: green dot
- Sizes of the dot change based on the distance between the player and objects. The further they are, the smaller the dots get.

To add another object to be detected, the object needs to have a Rigidbody, Collider, and a 'RadarElement.cs'. It is possible to add them through an inspector once the object is selected. Additionally, they need to have either a 'target' or 'pickups' tag to have different dots on the screen. The selection for tags is in the inspector under the name.

Reflection

During the whole process, I tried to iterate on an existing script from start to finish. I am not sure if it was a good idea but I think it was better than starting completely from scratch. I just had a vague idea of how to design the Radar System and then merged it together along with the compass script. I felt a bit guilty using a pre-made code but I did have a chance to understand how they used vectors to calculate the location and how dictionaries work in Unity.

In the beginning, I tried to design the case before looking at the code, however, it did not feel very comfortable to me so the design is very vague. After skimming through tutorials on how others tried to program Radar Systems using raycasts, I did think it would be useful to do it the same way as them. However, my experience with raycasts was not the best, and every time the game glitched as I relied heavily on it. Therefore, I tried to use box colliders for the Radar System instead. After a few of my sketched plans for the case did I look at the project. Only when I looked at the project, I had a clearer idea of what I needed. By then, it felt like using the compass script from the project was the best option as I just needed to add a few changes. I was conflicted about how Assembly definitions and namespaces work. Even though I did try to look through the Unity Manual I had a hard time understanding how they exactly work.

One regret for me was that I couldn't implement shaders. Even after watching numerous YouTube tutorials, I barely understood part of the basics of how to code Unlit Shaders. With Shader graphs, it might have been easier but I thought it would be great to know how to code shaders from scripts

instead. I cannot say if I tried hard enough but I do plan to look into shaders every block as they fascinate me that they can add great effects.

In the end, I did learn a few more concepts from programming, revised data structures, and had to work with vectors. When I include my goals for the Focus Track, I must say I am partially satisfied with my results. Although I believe there is a lot to improve especially regarding optimizing the code. As much as I don't fancy programming, I still love doing it.

References

Ameye, A. (n.d.). *5 ways to draw an outline*. Retrieved from https://alexanderameye.github.io/notes/rendering-outlines/

Documentation, U. (n.d.). *ShaderLab*. Retrieved from Unity Documentation: https://docs.unity3d.com/Manual/SL-Reference.html

Fandom. (n.d.). *Motion Tracker (Arious)*. Retrieved from https://avp.fandom.com/wiki/Motion_Tracker_(Arious)

Learn, U. (n.d.). *FPS MicroGame*. Retrieved from UnityLearn: https://learn.unity.com/project/fps-template

Madalaski. (2020, November 7). *INTRO TO SHADER PROGRAMMING - UNITY SHADERLAB*. Retrieved from YouTube: https://www.youtube.com/watch?v=SPKDjHkLnY4&list=PLkDXdQd1IwCMWkqBTt6jaaVxNv5TPobUr&index=40

Monkey, C. (2019, November 1). *Awesome Radar Effect in Unity!* Retrieved from YouTube: https://www.youtube.com/watch?v=J0gmrgpx6gk&t=1039s

Nolet, C. (2022, March 7). *Quick Outline*. Retrieved from Unity Asset Store: https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488

Taranovskii, A. (2020, Nomber 25). *10 Steps To Create Radar in Unity*. Retrieved from YouTube: https://www.youtube.com/watch?v=AV2AqjGl66o&t=1s

Wiki, W. T. (n.d.). *War Thunder Wiki*. Retrieved from https://wiki.warthunder.com/SPAA_radars