B2

Analysis of Brief

Analysis of Brief

The brief has asked me to create a videogame, digital animation or a physical game based on the prompt: "The Impact of A.I." As well as this, I must write a production log that includes in-depth breakdowns of idea generation, design, the development process, and the bug fixing process. There will be 2 presentations, one pitching the game idea and the second at the end of production. The brief has told me that the target audience is 16-25 year olds. According to

, this age range typically likes more violent and serious games, such as first/third person shooters and games with in-depth mechanics such as RPGs and competitive games. I must explore existing work that is similar to the prompt in order to inform my ideas and plan a better outcome. I then need to be creative and ideate on my design idea to refine it and create a better end result. Another important part of this project is management. I need to create a plan that includes timing for each part of the assignment, so that I can stay on track and making the right amount of progress.

Research

Game Research

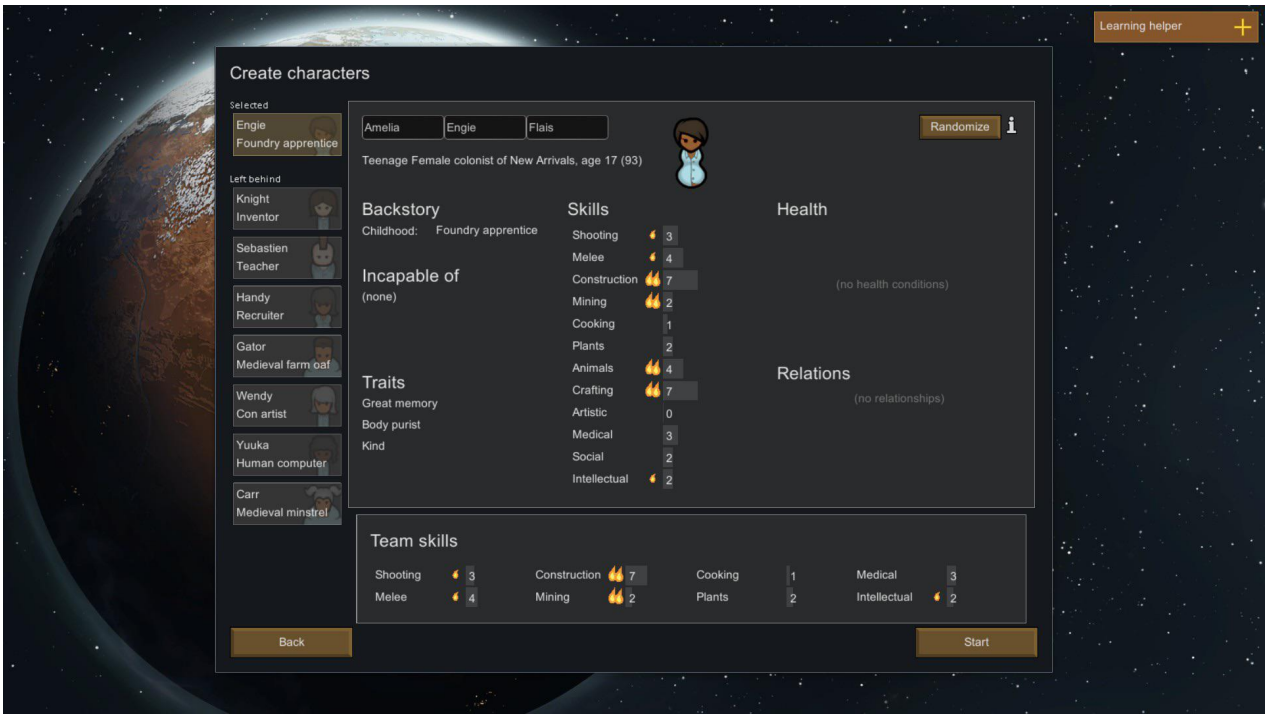
F.E.A.R.

An early example of great AI is the first-person shooter F.E.A.R, in which the developers created enemy AI NPCs that will react to the environment it is in. For example, if it senses it is in danger, it may fall back to a further piece of cover, or if it detects that there is an opening, it may even flank the player and attack from behind.

This adds a sense of realism to all the fights, as the enemies are unpredictable and act like real soldiers might. It creates much more dynamic encounters and boosts the replay value. F.E.A.R. is the gold standard for enemy AI in shooters and its design decisions such as Goal Oriented Action Plans and Finite State Machines are still used in modern games such as Metro Exodus (2019) and Half-Life: Alyx (2020).

Rimworld

Another example of good NPC AI in games is Rimworld. Rimworld is quite different to F.E.A.R., as it is a top-down 2D base building game in which you partially control multiple characters at once. At the beginning of the game, you choose 3 characters from a preset list, and each of these characters has unique skills and traits.



For example, a character could be very good at cooking, but bad at construction and

shooting. They can also have traits such as "Smoker", which means they need to have access to cigarettes, or "Hates Animals," meaning they receive a mood debuff when around animals. In-game, your characters also have needs such as food, sleep, recreation and mood. Depending on the events in the game, their mood may decrease, for example if they are sleeping in a cold room or get injured. Mood can also increase with events such as being social, eating a good meal, or winning a fight.

Since you can't control multiple characters at once easily, the characters in Rimworld have a large degree of automation and AI control. You can manually set the order in which different people complete tasks by using the "Work" screen.

For example, if a character spots a fire, and it is currently doing a task with a priority of 2, it will quit that task and automatically put out the fire, as firefighting has a higher priority of 1. This allows you to automate the game much more and create an efficient colony with the help of AI. This is similar to F.E.A.R.'s Goal Oriented Action Plan, as it selects a goal for the character based on the current environment and conditions. For example, if the food in storage is low, the person with the highest priority for cooking will start cooking meals automatically.

S.T.A.L.K.E.R.

The S.T.A.L.K.E.R. series has a very interesting approach to AI enemies, as it has 10 different factions that inhabit the game's open world. For example, if you are part of the Military faction, the Clear Sky faction will be hostile, but you are friendly with the Ecologists. You can start as any faction, meaning every playthrough is unique as you will be fighting a different set of enemies.

The AI system in S.T.A.L.K.E.R. is called A-Life, and it is designed using GOAP as well. Any AI that is within 150m of you is considered "Online," meaning that it is active and can move around/shoot, etc. Each "Online" AI has its own goals and routines for the day. For example, someone in the Loner faction may wake up at 8am, hang around their camp for a while, then squad up with some friends to go to to a mutant hideout and hunt some mutants for food. Then they may cook and eat, and go to an Anomalous Zone to hunt valuable Artefacts before returning to camp to relax by the campfire until night. Even enemies engage in this behaviour, meaning its possible to find an enemy squad roaming the open world, not just at their bases. This is called Emergent Gameplay, where multiple procedural systems interact with each other in different ways and give way to unique gameplay that is different for every playthrough. You can also interact with every friendly AI by walking up to them. You can usually get a quest from them, such as finding a specific gun and giving it to them, or clearing out a mutant/enemy camp. Some AI's are also dealers, meaning you can buy and sell items from them. For a game released in 2009, A-Life was quite an ambitious AI system. It had quite a few glitches, for example the AI characters could actually complete the main quest before the player and finish the game. In S.T.A.L.K.E.R. Anomaly, a fan-made mod pack that expands the game, these relationships are also dynamic. For example, if you finish a gunfight and there is an enemy lying on the ground that isn't dead, you can choose to either heal them, leave them to die, or kill them. All of these choices will impact your standing with that faction in different ways.

AI Research

AI is a very broad subject of computer science, and it has been portrayed in many different ways throughout its conception. In 1984, one of the first books to tackle the subject of AI, Neuromancer was released. It explored the possibility of storing people's consciousness in computer storage and being able to create new ones, as well as

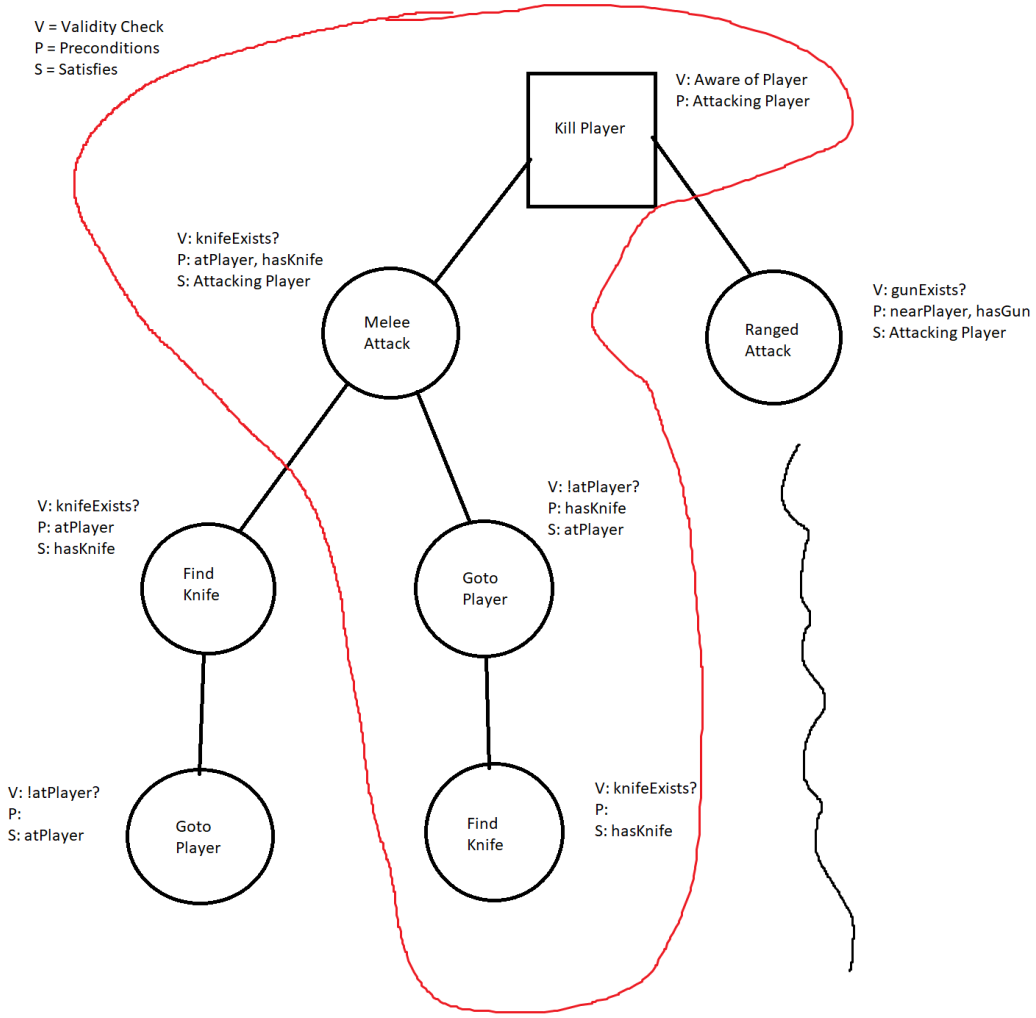
putting those consciousnesses into robots. It invented the phrase "Matrix" to describe a simulated world that lives inside a computer. This being one of the first pieces of media to be in a Cyberpunk style, it influenced many future films, games, and other pieces of media such as Cyberpunk 2077, The Matrix and Ghost in the Shell. An example of an AI character is GLaDOS from the Portal games. She is the AI controller of Aperture Science, a large scientific research facility that the Portal games take place in. She oversees your gameplay, adding context and humour as you complete the game's tests, and by the end of Portal 2, becomes a character you can empathise with and understand.

These are examples of AI as a narrative concept, however in the context of games, AI can mean a different thing. The Non-Player Characters (NPCs) that fill up many games use code to make the player feel like they are interacting with a realistic character. This form of AI uses many different techniques to achieve this, for example pathfinding to make the NPC move to the correct position, or advanced algorithms to decide how an NPC may act in combat as well as choosing the right voice line to play when an event occurs.

Goal Oriented Action Planning (GOAP)

GOAP is an AI system that was first used in F.E.A.R. and has been used in many games since. It is a very robust and "smart" system, meaning that there is a lot of nuance in the ways that NPCs can act in response to the player, and they can even do things proactively, without any input from the player.
It works by setting an end goal, usually something like "Gain Money," or "Kill Player," and then there are multiple actions that can lead to that outcome. Each of these actions has pre-conditions (for example, you need a weapon in order to hunt for food).
For example, in a game where the AI NPC's goal is to eat food, it may be able to choose between foraging for berries or go hunting for an animal.
You can also provide costs for each action plan, meaning that if an NPC already has a weapon to hunt with, it may prefer to go hunting, however if it needs to create a weapon and then go hunting, it may decide to forage for berries instead, as it is a simpler action plan.
This system is very good and allows NPCs to act in a much more immersive and realistic way, properly choosing the correct action in order to achieve their goals. However, it is mainly useful in games such as Rimworld and other survival games. This is because the system is quite overkill for FPS enemies, as they don't have many actions that they can

perform aside from shooting, reloading and moving.



Finite State Machines (FSM)

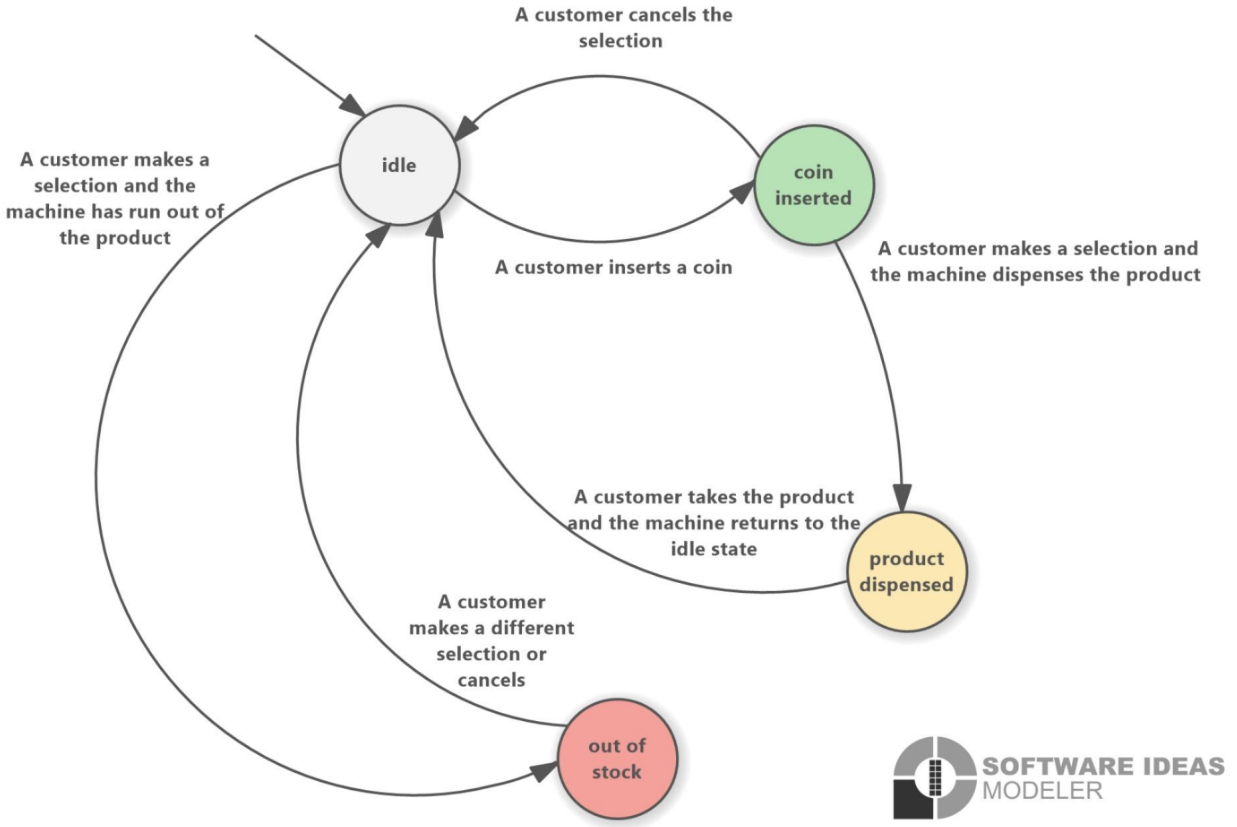
An FSM is a simple bit of logic that is used in many different systems, including engineering, software development, and game development.

It works by creating multiple "Nodes" or "States", with transitions between those states. For example, a few States for an enemy AI could be things such as "Attack," "Idle," "Patrol," "Push," or "Flee." They are essentially what the AI is currently doing.

The AI starts in an "Idle" State, and once it spots a player, it may transition into the "Attack," State. Then, after it has fired a few shots it may transition into a "Flee," or "Push," State, depending on many different factors such as if it hit any shots, if it is damaged at all, if it has teammates nearby, etc.

However, FSMs are not very scalable. Once you start adding many different States, it becomes more and more complicated to handle the transitions between every node, increasing development and debugging time for more complex State Machines. This means that it is a good system for fairly simple AI's, such as those in platformers like

Mario, or casual shooters such as Risk of Rain 2.



Behaviour Trees

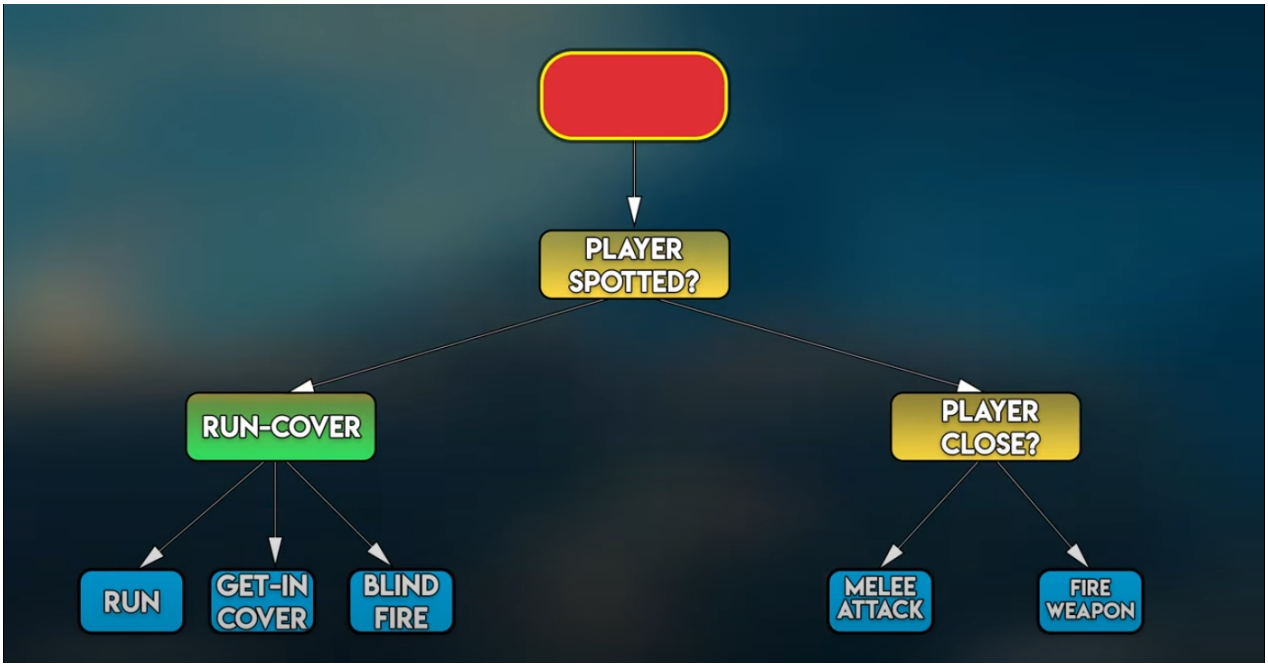
Behaviour trees are quickly becoming one of the most used AI systems in AAA games. They work in a similar way to GOAP, however different in a few very important ways. Instead of having an end goal, the flow of the tree starts from the root node at the top, and the current behaviour is decided by moving down through "decision" or "selector" nodes. A selector node could be something like "Is player within a radius," or "Is out of ammo."

Depending on the result of these selector nodes, the current state of the AI NPC will be changed. For example, if the magazine is below 5 rounds, the selector node may change the state to reloading instead of shooting.

Behaviour trees have many upsides when compared to an FSM, specifically you don't have to individually handle every transition between states as the tree does that automatically. As well as this, it is much easier to debug as you can view the current path that is being taken in a simple graphical display. This makes it much more scalable.

This is also the only system that doesn't require any coding experience, so it is much

more accessible to designers.



Ideation

Ideation

I knew that I wanted to create a first-person shooter, as it is a great way to experiment with hostile AI and a lot of my favourite games are FPS's. However, FPS is a very broad genre and has many different styles and speeds of gameplay. For example, Escape from Tarkov is a very slow, tactical and realistic shooter with accurate bullet and armour physics, meaning one shot is enough to kill in many circumstances. In comparison, a game like Titanfall 2 is very movement-based and fast-paced, with a long time-to-kill (TTK). Titanfall 2 has advanced movement mechanics such as double-jumping, sliding, wall-running and bunny-hopping which pushes the skill ceiling very high.

Idea 1: Tactical Shooter

My Initial Idea was to create a Tactical Shooter in the style of Escape from Tarkov and Counter-Strike 2. These games have a very high skill ceiling and play quite slowly, as they have slow movement speed and increased inaccuracy when walking. This allows for a very tactical game with a lot of high-importance decision making and many ways to approach a given situation. In order to fit the brief, I would implement enemy AI and potentially a friendly AI that you can direct to open doors, throw flashbangs, as well as covering your rear for any flankers. I would need to code a custom AI to follow your commands, as well as staying out of your way and not blocking any doorways/passages. As this game would be single-player, I think a good game loop could be inspired by the Armed Assault (ArmA) series. These games have multiple "scenarios," which you can complete in any order you wish. An example of a scenario may be travelling to a town that is enemy occupied, attacking the town, destroying radio equipment or vehicles etc, and then exfiltrating the area. I think this is a good game loop to bring back, as it is very

open and allows the player to approach the current situation in many different ways. As well as this, it may be possible to randomise enemy spawn locations and let you infiltrate the area from multiple entry points. I think this would increase the replayability of each scenario, and encourage speedrunners to play the game. I am inspired by Eastern European shooters such as Tarkov and S.T.A.L.K.E.R. for the visual style, as they are both quite realistic and grounded.

Idea 2: Movement Shooter

Another idea I had was a very fast, movement-based shooter with grappling hooks. I was inspired by the game Titanfall 2, as well as the Mirror's Edge games to create a shooter with a lot of map traversal and verticality, as well as being able to hit people without needing to aim down sights, or stay still to maintain accuracy. It could also score you based on time as well as bullets used, allowing sections to be replayed in hopes of "speedrunning" the areas. This would increase the replayability of the game and might attract speedrunners. Large maps with semi-non-linear pathways could make it more interesting to find new paths through the map for a faster time. The visual style for this game would likely be futuristic/sci-fi, as advanced movement mechanics in a realistic or historical setting may be jarring or out of place. The narrative is that you are part of a group of the last humans in the world, after the rest have been killed by a rogue AI. The story of the game is being hunted by a powerful AI that sends robots out to kill you.

Idea 3: Realistic Sniper Game

Another game idea I had was a

- Realistic Zeroing and Bullet physics
- Bipod stabilisation
- Bolt action rifles
- Simpler AI, will try to locate you and fire back as well as running to cover if they spot you.

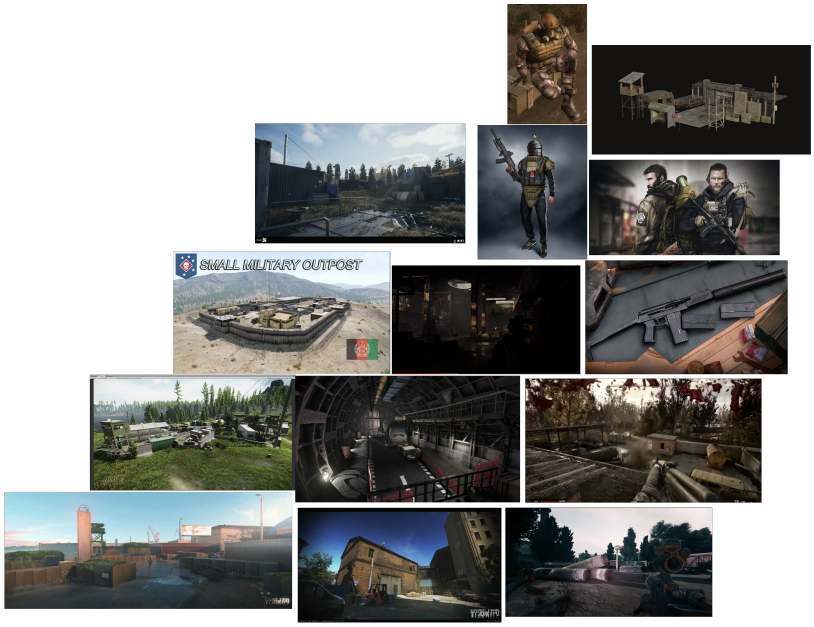
Idea 4: Base-builder

Inspired by Rimworld, Foxhole and Dwarf Fortress Set in the post-apocalypse, you are tasked with rebuilding a city using many AI NPCs. Top-down perspective, you micromanage hundreds of people with their own jobs such as collecting wood, mining resources, building structures, etc. You also need to balance the number of people in your military. More people in your army means fights will be easier, but there are less people to do normal jobs. Maybe an XCOM or Red Alert-style military system, with turn based combat and managing the entire army.

Idea 5:

Chosen Idea

I decided that I would choose the first idea, a tactical shooter. I chose this because I have more experience in similar games and they are some of my favourites to play. Realistic games have also always been very interesting to me, as they have the ability to immerse you further in the world through mechanics such as permadeath, realistic health and bullet mechanics and punishing AI. I created a moodboard to help pin down the visual style and setting of my game. I want to have a modern, Eastern-European setting, with a large military presence and realistic area maps.



I then started to think of the narrative and setting of the game. I want a realistic conflict that could happen in real life, as well as taking place in a fictional country to avoid real life parallels. I named the country Sujusterea, and it is located along the Baltic sea, neighbouring Latvia and Lithuania. A civil war has been brewing for a long time, since the previous one in 1968 and the government's increasing allyship with Russia has angered the citizens.

As well as this, I considered which game engine to use for this project. The main 2 I considered were Unreal Engine 5 and Unity. Each engine has its upsides and downsides, for example, Unreal Engine 5 has access to Quixel Megascans. This is one of the largest 3D asset databases in the world, including over 18,000 photorealistic and photo-scanned assets for completely free, as long as you use them inside Unreal Engine 5. In comparison, Unity has a few free assets on the Unity Asset Store, however they are made by the community and have varying levels of quality. Unity uses C# as a scripting language, compared to C++ in Unreal 5. Both have visual scripting solutions, however I prefer to use code to node-based scripting. I am much more familiar with Unity, and have

been using it for around 3 years now, and have only slightly dabbled with using Unreal 5. For this reason, I will use Unity as it allows me to realise my ideas much more quickly as I have a better idea of how to use the program.

Game Design Document

Game Design Document

Overview

Sujusterea: Civil War is a Single-Player or Co-op Realistic Tactical Shooter in which you play as part of an insurgent group, fighting against the Sujusterean Military in a fictional modern-day civil war.

The game takes place in sessions where you decide an infiltration location, and then retake Forward-Operating Bases (FOBs) by infiltrating the location and clearing it of enemies. There are multiple side-objectives to each infiltration, such as picking up intelligence, freeing prisoners etc. Once you have completed the objectives that you can, you then proceed to an exfiltration area (on the other side of the map from your spawn).

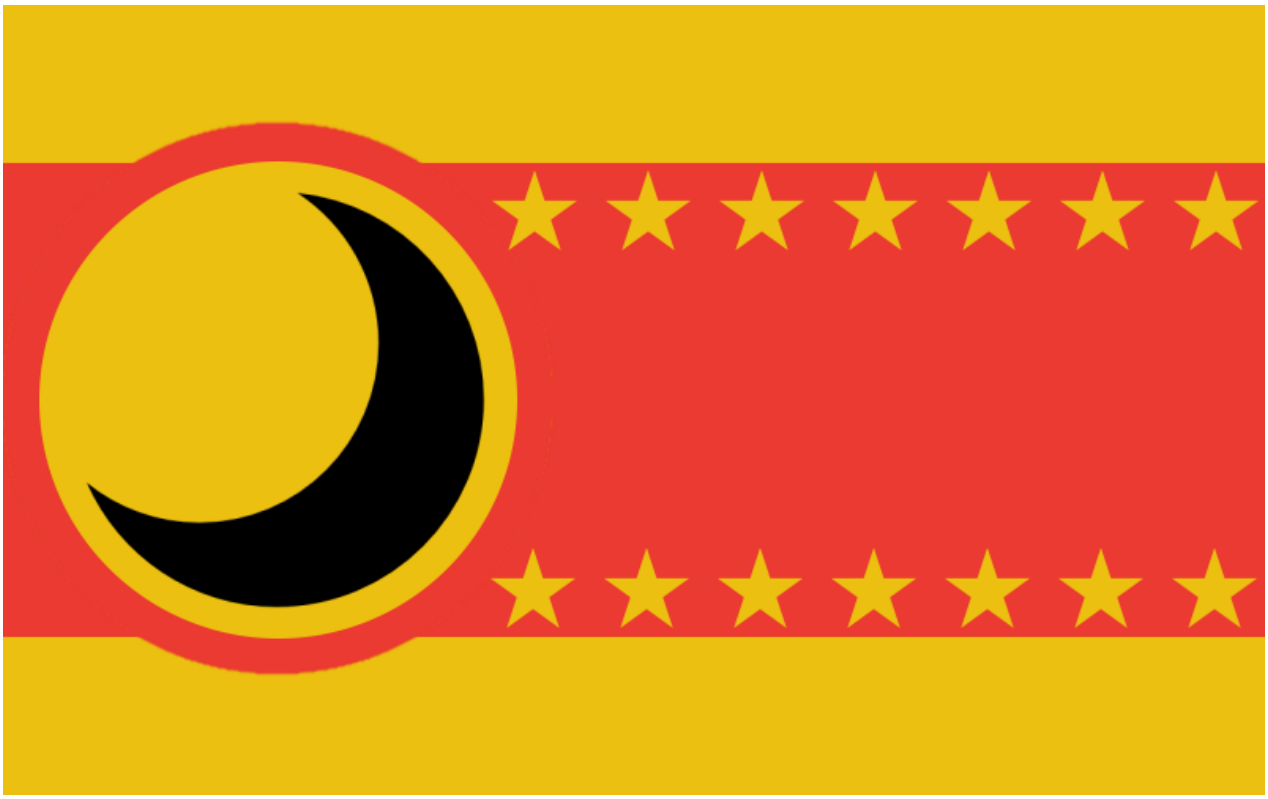
You play against multiple AI enemies that take defensive positions inside the map, and you have an AI teammate that you can direct to suppress fire, watch your back, or stay back and keep quiet for stealth segments.

Controls and Player

Since the game is intended to be realistic, I want to approximate a real human's capabilities with the player controller. This means mechanics like a relatively slow base movement speed, weapon sway and inaccuracy while walking (so you need to stay still to shoot accurately), as well as the ability to lean around corners and slow walk for silent/quiet movement. This adds a lot of complexity and depth to the movement mechanics, for example the enemies need to consider if they are close enough that they can hear you. I aim for the gameplay to be relatively slow and tactical. Additional controls such as crouching and laying prone are also planned, as it makes sniping much more viable as a tactic. Taking inspiration from the ArmA series, I decided to add a feature where you can hold down the right click button to zoom the camera in by 1.5x. This is a fun game mechanic as it allows you to survey the area in much more detail than you could with a higher Field of View (FOV), and potentially spot enemies further away. However, I think it would be a bit too powerful if you could aim down sights as well as zoom, especially with a magnified optic, so you are only able to use the ArmA-style zoom if you are not aiming down the sights.

Theme/Setting

It is set in the fictional Eastern-European country of Sujusterea (located along the Baltic sea, neighbouring Latvia and Lithuania), in which the current government is starting to ally with Russia. Historically, during the USSR, Russia has oppressed Sujusterea, and the citizens now believe that the politicians at the top of the Sujusterean Government are accepting bribes from Russia. Picking up any weapons they can find, and quietly aided by NATO and the US, the Force of Operations for the Republic of Sujusterea (FORS), led by Sebastianas Forsentaciuki will rise up against the Sujusterean Military Forces (SMF).



Visuals

I aim for the visuals to be as realistic as possible, however as this will just be a game prototype, I will use free assets I find online using platforms such as the Unity Asset Store and Sketchfab to fill out the game world. This means that the game will likely look quite simple and there will be many assets with a different art style.

Game Loop

Once you choose a map from the main menu screen, it shows the Infiltration Screen which includes the functionality to select where you are infiltrating from, selection for the guns that you want to bring in, selection of your teammates and the current objectives + objective locations. Once you have selected all of that, you can infiltrate. Then, you spawn in with your teammate(s) and begin attacking the objectives. Objectives may be things like clearing an area of enemies, picking up a piece of intelligence, Once you have completed all the objectives that you want, you head to the exfiltration zone and exit the map.

AI

There are 2 types of AI that will be in the game- Enemy and Friendly.

Enemy AI will be more defensive, holding positions in cover and waiting for your attack to retaliate. They will spawn in different pre-set places randomly, so that each play-through will play differently and replay value is increased. If you fire on an enemy that does not know you're there (stealth), then they may retreat and find a piece of cover to hide from you, or if they have the jump on you (I.E. they can hear you coming close), then they can push you aggressively and maybe throw a grenade. The likelihood to perform these actions are influenced by multiple variables, such as how many friendly AIs are nearby, current enemy health, if they just hit a shot on you, if they have low ammo, etc.

The other type of AI will be friendly teammates that support you and can be indirectly controlled using an interface you can open with the middle mouse button. You can command teammates to:

- Open suppressing fire on a specific area to keep enemies heads down
- Open and flash a door for you
- Watching your back
- Toggle stealth/standing mode

Each AI teammate has a unique name, and you can hire new teammates with different skill levels as you progress through the game. AI teammates can also get shot and you may need to heal/help them up if they are downed. They also have perma-death, meaning if they are downed for too long and die, you cannot revive them or restore a save. This is intended to give you a more personal connection to the teammates that you play alongside, as you may miss a particularly high-skilled teammate if they die in combat. This is where I think the prompt is reflected the most. I aim to code my own AI system using Unity's Navmesh system, which is something I have not attempted before.

When deciding which AI system to use, I looked back at the research I did earlier. Each system has upsides and downsides, and specific use cases that would be best suited for. I ruled out GOAP, mainly because my enemies have very few actions they can perform. There aren't any actions that have to be sequenced either, only shooting, moving and reloading, so I would not be making advantage the main upside of GOAP. It has a much more in-depth and complicated implementation procedure, so would take longer to debug and perfect.
Behaviour trees would be a suitable solution to use, and I prefer it to FSM as it is much easier to understand and debug, as well as not needing any code. As well as this, it is one of the most used systems for AI in modern games, and it would be good to have an understanding of how to create them for the future. However, unlike Unreal 5, Unity does not have a built-in behaviour tree system. In order to make one, I would need to purchase a user-made one from the Asset Store. As I don't have a budget for this game,

I decided to use an FSM instead.
While this makes it harder to implement, I think it is good practice for coding and allows me to learn a simple design pattern that can be used in many different scenarios, even outside of game design in general.
As the AI will be relatively simple, an FSM shouldn't be too hard to implement. # Level Design Taking inspiration from games like Escape from Tarkov and S.T.A.L.K.E.R., the game takes place in multiple large areas of around 500m^2 to 1000m^2. Each map will include forested areas, as well as locations such as towns, industrial parks, military bases, factories, etc. Each area will have 2-4 infiltration points, where you can choose to start the map at a different location. This adds replay value as it allows for a lot of player freedom and expression. For example, a location might be more heavily guarded towards the north side, so you can choose to sneak in through the south and take them out from behind. At any point, you can head to an extraction zone and exfiltrate the area, ending that game, however the objectives you did not complete will fail. # Health / Death Enemies will die in one hit to the head, or 2-4 shots in the body. They can sit in place and slowly heal, however are vulnerable to pushes while this is happening as their gun is not ready. The player's health system is similar, in the way that you can heal by putting your gun away. As well as this, If you get killed, you enter a "coma" state, in which your teammate must rush to you and give you aid, or you will die in 90 seconds. Your teammate can enter the same "coma" state, and you will need to help them up. # HUD and Menus I want the UI to be minimal and clean, only showing information when it is necessary. For example, the ammo indicator will only show while you are shooting or reloading the gun. There will be a crosshair on screen, using a raycast from the barrel of the gun to determine its end position. This means that when weapon sway from walking, breathing, moving etc. is added, it also affects the position of the crosshair, so it is always accurate to where the bullet will go. This is the way I will add moving inaccuracy. I have also make a mockup of the Infiltration Screen and the UI for commanding friendly NPCs. # Audio I intend to add audio to my game, again making it as realistic as possible. I also want AI enemies to be able to hear you, and react even if you are on the other side of a wall.

Project Management

Project Management

I had to plan out every feature I needed to add in order to stay on track and make sure I have enough time to implement every feature. I decided to use a calendar instead of a Trello board which I usually use. I decided to try something different as I didn't like Trello that much. Having a direct link to time is very useful as I can see exactly when I should be working on specific targets.

A very important part of project management in all forms of software development is using Source Control, also known as Version Control. This is the practice, usually through a third-party application, of logging/tracking every change you have made to the

source code and assets across development.
There are many different ways to handle version control, for example Git + Github, Unity's Version control, and AWS CodeCommit. They are all fairly similar, however I chose to use Github as it is what I have used in the past and is considered the Industry Standard.
Version control works by logging every change you make to a script, as well as every asset you add to the repository and stores it in a file called a "commit". You can then push that commit to Github's servers, along with a small description of what you have changed, and anyone that has access to that repository can download that commit onto their computer and stay up-to-date.
This was extremely useful for working on multiple machines, as I can sync the computers from the Github cloud anywhere. Version Control also allows you to check out old versions of your repositories, meaning you can roll back changes that are bad, or view previous versions of the repository. ![[Pasted image 20240426155551.png]] ![[Pasted image 20240426155634.png]] ![[Pasted image 20240426155651.png]]

Devlog

Devlog

Character Controller: Feb 20th - Feb 26th

The first thing I had to do was to create a first-person character controller. I need to be able to walk, crouch, sprint, jump, lean and potentially lie prone.

After doing some searching online, I found an example Github Repo that mimicks the Source Engine movement code. This was the code used in CS:2, Half-Life 2 and other Valve games. I also found some short videos that explained this code in very simple terms, and helped me bugfix. I read through the movement code, and translated the parts I needed into my own project. I missed out on things such as ladder movement and swimming, as the map doesnt include either. The camera and player movement happens in the `Player.cs` file.

```
private void CalculateGroundMovement()
{
        //wishDir is direction you wish to move in, or the vector directly resulting from your key presses
    Vector3 wishDir = Vector3.Normalize(inputMovement.y * transform.forward + inputMovement.x * transform.right);

    float wishSpeed = maxSpeed / 100;

    //currentSpeed is not actually the current speed, but the dot product of the current velocity and the direction your arrow keys are
```

```
pressing.
    //This is what enables air-strafing, allowing the player to
bunnyhop.
    float currentSpeed = Vector3.Dot(velocity, wishDir);

    float addSpeed = wishSpeed - currentSpeed;
    float accelSpeed = Mathf.Min(accel * Time.deltaTime * wishSpeed,
addSpeed);

    //Add speed in wish direction
    velocity.x += accelSpeed * wishDir.x;
    velocity.z += accelSpeed * wishDir.z;

        //Calculate the deceleration
    float speed = velocity.magnitude;
    float drop = speed * decel * Time.deltaTime;
    float newSpeed = Mathf.Max(speed - drop, 0);
    if (speed > 0) newSpeed /= speed;
    //Apply deceleration
    velocity.x *= newSpeed;
    velocity.z *= newSpeed;
}
```

This creates a much higher quality character movement controller than a lot of other solutions for Unity, as it has smooth acceleration and deceleration.

```
    private void CalculateView()
    {
            //Delta input for mouse movement
        inputView *= sensitivity / 100;

        cameraAngles += inputView;
        //Clamp the Y axis to +-90 so you can't do "backflips"
        cameraAngles.y = Mathf.Clamp(cameraAngles.y, -90, 90);
                //Rotate the entire player on the Y axis, and only the
camera holder on the X axis.This means the entire player object is
always looking forward, but the player does not glitch out when you look
up or down as vertical rotation is only done on the camera.
        Quaternion camRot = Quaternion.AngleAxis(-cameraAngles.y,
Vector3.right);
        Quaternion playerRot = Quaternion.AngleAxis(cameraAngles.x,
Vector3.up);

        camHolder.localRotation = camRot;
        transform.localRotation = playerRot;
    }
```

Weapon Sway: Feb 27th - Mar 3rd

Once I had completed the movement controller, it was time to work on the weapon controller. I searched online for some tutorials on first person weapons in Unity, and found this very helpful tutorial series.

First, I worked on weapon sway, which is the gun's movement when you look around quickly, are currently walking, etc.

```
void CalculateWeaponRot()
{
        //if you are aiming down sights, scale each component of the
sway down by individual amounts
    float _movementScaler = 1;
    float _swayScaler = 1;
    if (isAiming)
    {
        _movementScaler = movementRotScaler;
        _swayScaler = swayRotScaler;
    }
        //Most weapon sway is done using Vector3.SmoothDamp
        //This smoothly moves the object to the target position, with a
smoothness set by the user
    weaponRotation.x += GameManager.GM.player.accumulatedInputView.y *
swayAmount;
    weaponRotation.y += -GameManager.GM.player.accumulatedInputView.x *
swayAmount;
    weaponRotation = Vector3.SmoothDamp(weaponRotation, Vector3.zero,
ref weaponRotationVelocity, swaySmoothing);
    newWeaponRotation = Vector3.SmoothDamp(newWeaponRotation,
weaponRotation, ref newWeaponRotationVelocity, swayResetSmoothing);
    newWeaponRotation.z = newWeaponRotation.y * 0.75f;

    movementRotation.z = -movementSwayAmount *
GameManager.GM.player.inputMovement.x;
    movementRotation = Vector3.SmoothDamp(movementRotation,
Vector3.zero, ref movementRotationVelocity, movementSwaySmoothing);
    newMovementRotation = Vector3.SmoothDamp(newMovementRotation,
movementRotation, ref newMovementRotationVelocity,
movementSwaySmoothing);
    //Apply the new positions multiplied by the aiming scaler
    wpnRot += newWeaponRotation * _swayScaler + newMovementRotation *
_movementScaler;
}
```

The tutorial series helped me with starting the weapon system, however it had many issues and I had to write quite a lot of my own code. For example, the youtuber used an animation clip for walking, but I wanted a more procedural system, where you can change the step speed and how far it steps side to side. It also allowed me to slightly randomise the step locations to add some realism.

```
void FixedUpdate(){
        //Simple iterator acts as a timer
        if (curWalkLifetime < walkLifetime * walkLifetimeScaler)
        {
            curWalkLifetime += 1;
        }
        else
        {
                //Swap foot
            curWalkLifetime = 0;
            rightFoot = !rightFoot;
        }
}
```

```
    void CalculateWalk()
    {
            //Scale down if aiming down sights
        float _walkScaler = 1;
        if (isAiming) _walkScaler = walkScaler;

                //if in the second half of the step, the target is the
rest position
        Vector3 target = Vector3.zero;
        if (GameManager.GM.player.velocity.magnitude > 0.01f)
        {
                //if you are in the first half of the step, target is
moved down and towards the foot that is currently stepping
            if (curWalkLifetime < (walkLifetime * walkLifetimeScaler) /
2)
            {
                float lateralVelocity = new
Vector2(GameManager.GM.player.velocity.x,
GameManager.GM.player.velocity.z).magnitude;
                target.y -= Random.Range(stepDownAmount.x,
stepDownAmount.y) * lateralVelocity;
                float sideAmount = Random.Range(stepSideAmount.x,
stepSideAmount.y);
                //Move to left or right depending on boolean set in
FixedUpdate()
                if (rightFoot) target.x += sideAmount * lateralVelocity;
                else target.x -= sideAmount * lateralVelocity;
            }
        }
        //Apply movement towards target
        walkMove = Vector3.SmoothDamp(walkMove, target, ref
walkMoveVelocity, walkMoveSmoothing);
        newWalkMove = Vector3.SmoothDamp(newWalkMove, walkMove, ref
newWalkMoveVelocity, walkMoveSmoothing);
```

```
        wpnPos += newWalkMove * _walkScaler;
        wpnRot += new Vector3(newWalkMove.y, newWalkMove.x, -
newWalkMove.x * 1.5f) * stepRotScaling;
    }
```

I also wrote a similar method for breathing, with some randomness to increase realism. Once I was done with that, I also needed to allow the player to aim down the sights of their gun. This is where I started to create the actual gun logic (shooting, reloading, recoil, etc.) Each firearm has 2 Vector3s called `restPos` and `aimPos`. These are the 2 positions of the gun at rest and while you are aiming down sights. They are different for each gun, so is stored in the `firearmInfo` class along with stats such as recoil, fire rate, fire modes and magazine size.

Weapons: Mar 4th - Mar 14th

I then added shooting mechanics.

```
void FixedUpdate(){
        //Single-Fire just calls the Shoot() method as soon as it is
pressed
    if (canShoot && fullAutoHeld && curFireMode == FireMode.fullAuto &&
!isReloading && roundsInMag > 0)
    {
        Shoot();
    }
}
 void Shoot()
 {
         //Instantiate round at barrel point
     GameObject roundObj = Instantiate(roundPrefab,
barrelPoint.position, transform.rotation);
     roundObj.GetComponent<Round>().firearmFiredFrom = this;

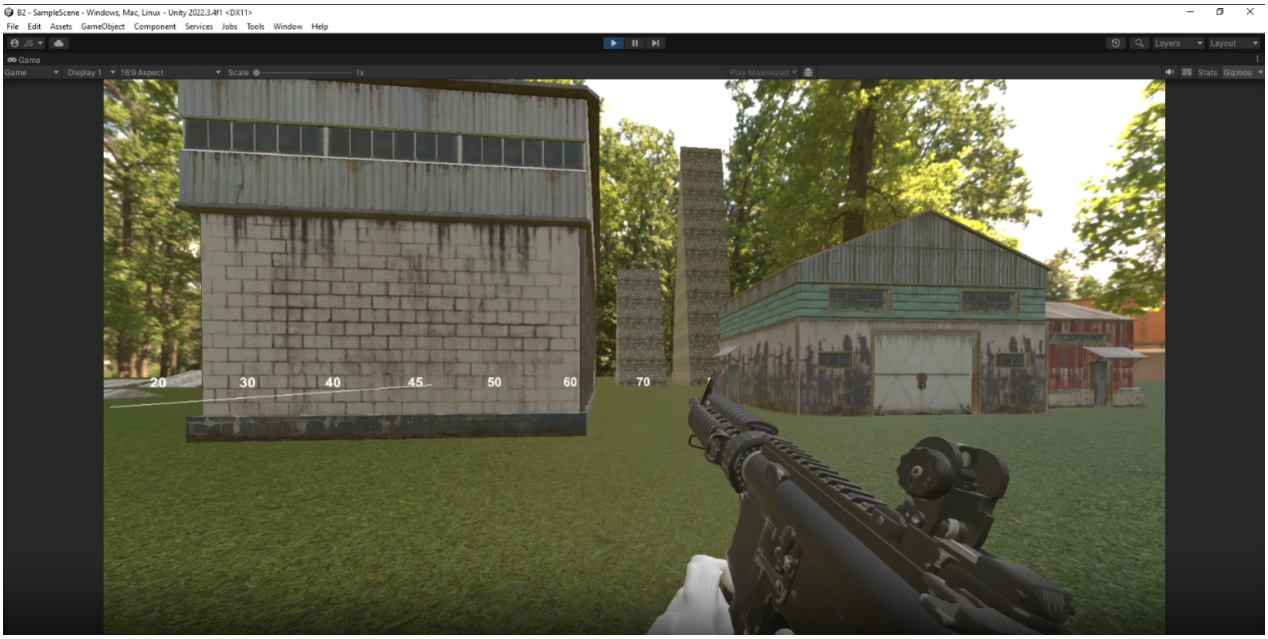
     AddRecoil();
     canShoot = false;
     roundsInMag -= 1;
     sustainedRecoilAdd += info.sustainedRecoilAdd;

     if (muzzleFlash != null)
     {
         muzzleFlash.transform.localEulerAngles =
new(muzzleFlash.transform.localEulerAngles.x,
muzzleFlash.transform.localEulerAngles.y, UnityEngine.Random.Range(0,
360));
         muzzleFlash.SetActive(true);
         Invoke(nameof(ResetMuzzleFlash), Time.deltaTime * 2.5f);
```

```
        }

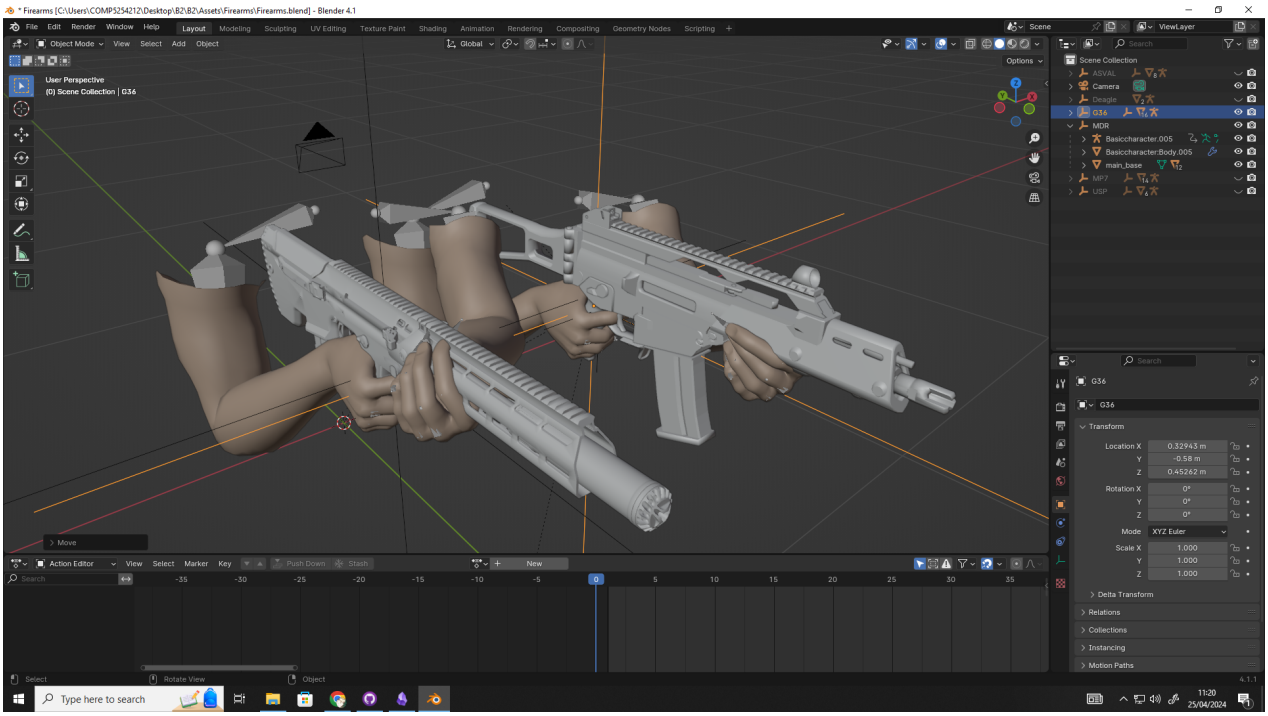
        if (roundsInMag ≥ 0)
        {
                //Invoke() calls a method after a set period of time
                //In this circumstance, ResetShot() sets canShoot back to
true after a single rpm interval (converted into seconds from rounds per
minute)
            Invoke(nameof(ResetShot), 1/ (info.roundsPerMinute/60));
        }
```

This is the first time I have used the `Invoke()` method, and it is a very useful tool for timing different actions. It can be used to time a full-auto gun to fire at the right RPM, as well as handling reloading. I also added a button (V by default), to change the fire mode from single fire to full-auto. Once all of the shooting was finished, recoil was next. Recoil is done in a very similar way to weapon sway, by adding rotation to the gun using `Vector3.SmoothDamp()`. There are 4 ways that recoil affects you, vertical rotation, horizontal rotation, camera rotation and lateral movement. Each shot's recoil is slightly randomized as well, to make spray patterns different every time.

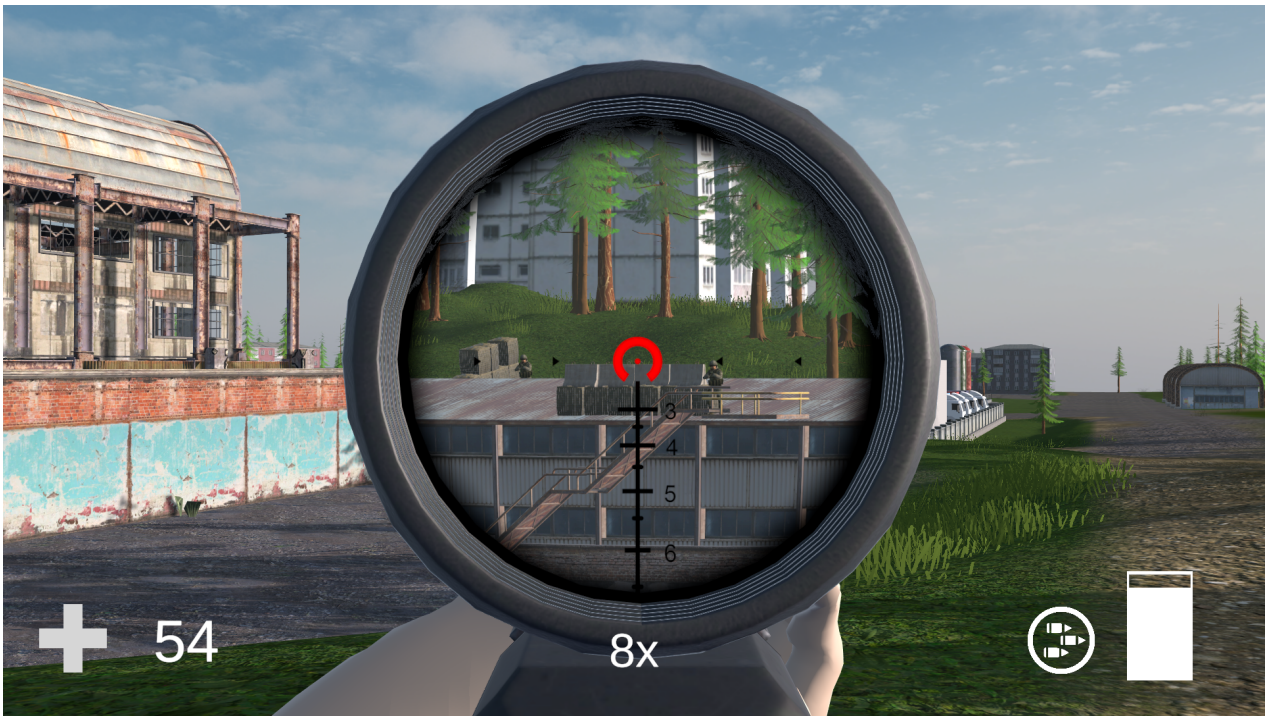


Mar 20th

I found 6 weapon models on Sketchfab, and began putting them into blender in order to get them ready for Unity. I had to manually position the hands, and added some very simple reload animations (your character moves the gun off screen then back on screen).



As well as this, I used 2 old shaders from when I created mods for H3VR, a red dot sight and a magnified optic. They were fairly simple to implement and I particularly like the optic shader, as it uses a `RenderTexture`. This is an additional camera that renders to a texture in real-time, and you can use that texture to display in shaders. This means that even though the scope zooms in, your peripheral vision stays at the normal FOV and you do not sacrifice spatial awareness when scoping in.



Map Design: Mar 14th - Apr 24th

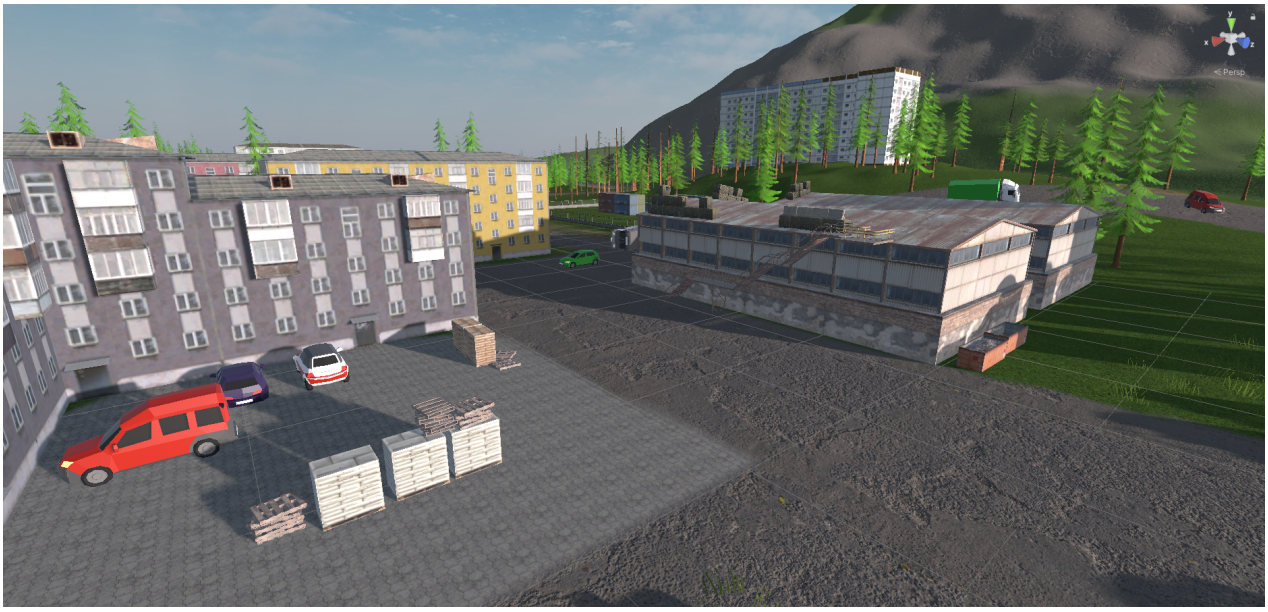
At this point, I knew that the debug scene I had created in order to test movement needed an update to further iterate on my game concept. I started making a larger scene that would become the first area in the game: "Clifftown". Originally a small town in Sujusterea, it was taken over by military forces and turned into a stronghold defending

important intelligence. With a large cliff overlooking the town, long range combat is inevitable, and you'll want to bring an optic.

Here is a very early photo of the map from the players perspective. I used many assets from the Unity Asset Store and the website Sketchfab, all of which were free and even though the visual style of my game is very kitbashed (meaning there are lots of different parts from different people, and the aesthetics are not consistent), I ended up with a functional and quite large play area for the initial build. I begun creating the map with Unity's Terrain system. This is a plane that you can deform and paint on to create the ground for the world.

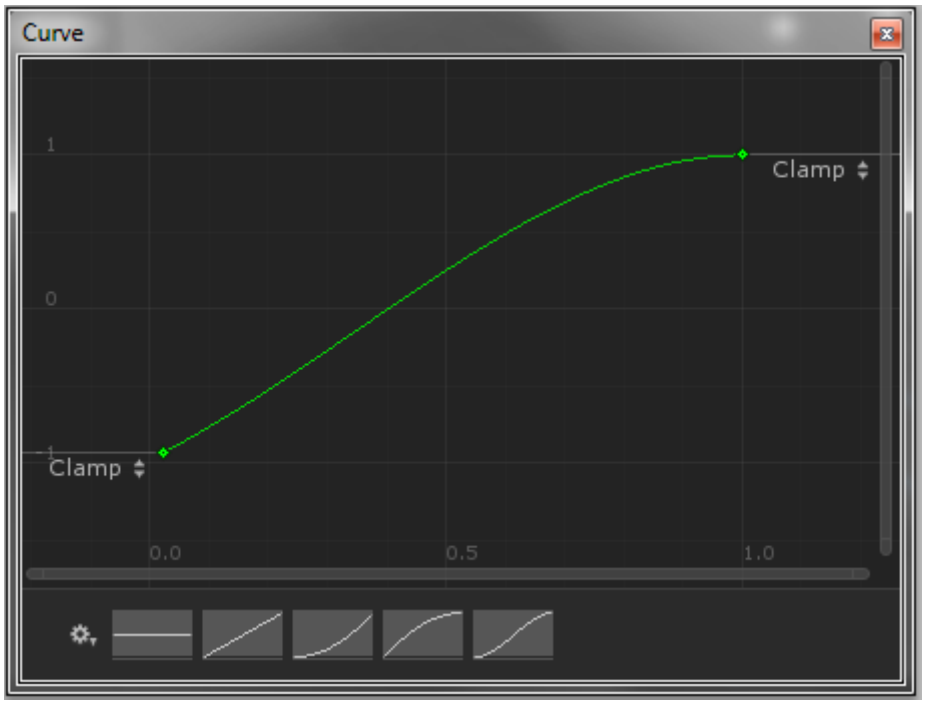
I then added pathways by painting gravel and dirt, added many assets, including buildings, military tents, tanks and watchtowers, as well as cover such as cars and trees. The grass and trees use Unity's terrain system as well, as there is functionality for

painting trees and grass onto the ground.



Bullet Physics: Mar 12 - Apr 11th

I knew that I wanted to have realistic bullet physics, including travel time and bullet drop. This actually was one of the hardest features to implement, as I ran into many issues with collision. The way that many games handle bullets is using a raycast. Raycasts are invisible lines that travel at infinite speed that you can send out and get information about the object that it hits. It is used everywhere in games programming, for example checking if there is a wall in-between the Enemy AI and the Player Character. They are a very simple solution for bullet physics, as they are very easy to implement and have very little performance impact. However, it has a few drawbacks, namely that they have to be in a straight line. This means that the bullet can not have drop-off at range, which is unrealistic. Another drawback is that all raycasts happen instantly, meaning the bullet travels at an infinite speed. This is not realistic and makes the gameplay shallower as you will not need to account for bullet drop and travel time when shooting at long distances. I had to use a new solution that treats the bullet as a physical object, instead of a raycast. Unity detects collision between objects using its collision system, and it checks physics updates every time a method called `FixedUpdate()` is called. This is typically 50 times a second, however it is possible to customise the time step inside Unity's Preferences. My first iteration of the bullet physics used a physical object that updated its position every `FixedUpdate()`. I also used a variable type called an `AnimationCurve`, that lets you put in an input value and receive a value from the curve. I used this to measure the bullet drop, and authored the curves to fit real world values for

each caliber. 

While testing however, many issues were apparent. Unity's collision system is notoriously bad at handling collisions between very fast moving objects. I have run into this problem before in some other game prototypes, where an object is moving so fast, it just phases through obstacles it should hit. Obviously, this is not good as bullets may go through enemies and walls.

```
    void FixedUpdate()
    {
        distFromOrigin = (transform.position - startPoint).magnitude;
        //Use distance from origin to evaluate animation curve
        float dropAmount = dropCurve.Evaluate(distFromOrigin);

        curPoint += (velocity.z * transform.forward)+(velocity.y *
transform.up);
        curPoint.y += dropAmount/1000;

        float velocityReduction = Mathf.InverseLerp(0, maxDist,
distFromOrigin);
        velocity.z *= velocityReduction / 100;

                //This should be done in the Start() method, however
this is old deprecated code
        Material newMat = new(tracerMat);
        newMat.color = tracerColor;
        newMat.SetColor("_EmissionColor", tracerColor);

        lineRenderer.material = newMat;


                //destroy for performance if too far away (2000m
default)
        if (distFromOrigin > despawnDist) Destroy(gameObject);
```

```
            positions.Add(curPoint);
            transform.position = curPoint;
        }
```

Apr 8

As this method had many issues with collision, I went online to find other solutions to my problem. I found a very helpful youtube tutorial that uses a hybrid method.

Every `FixedUpdate()`, a new raycast is sent, going from the current point to the "next point." The next point is calculated in the `PointOnParabola()` method.

```
    private void Start()
    {

        startPosition = transform.position;
        startDir = transform.forward.normalized;

        lineRenderer.enabled = false;
        Invoke("Show", 0.05f);

            //Set up color and materal of line renderer for tracers
        Material newMat = new(tracerMat);
        newMat.color = tracerColor;
        newMat.SetColor("_EmissionColor", tracerColor);
        lineRenderer.material = newMat;
    }
    void Show()
    {
        lineRenderer.enabled = true;
    }

    Vector3 PointOnParabola(float time)
    {
        Vector3 pos = startPosition + (muzzleVelocity * time * startDir);
        Vector3 gravityVector = Vector3.down *(gravity * time * time);
        return pos + gravityVector;
    }
    bool RayBetweenPoints(Vector3 startPoint, Vector3 endPoint, out
RaycastHit hit)
    {
            //Do raycast from current point to next point
        return Physics.Raycast(startPoint, endPoint - startPoint, out hit,
(endPoint - startPoint).magnitude);
    }
    private void Update()
    {
        if (startTime < 0) return;
```

```
        float currentTime = Time.time - startTime;
        Vector3 currentPoint = PointOnParabola(currentTime);

        transform.position = currentPoint;
    }
    void FixedUpdate()
    {
        distFromOrigin = (transform.position - startPosition).magnitude;

        if (startTime < 0) startTime = Time.time;
        float currentTime = Time.time - startTime;
        //fixedDeltaTime is the amount of time that one FixedUpdate takes up, so we are calculating what point the currentPoint will be next FixedUpdate
        float nextTime = currentTime + Time.fixedDeltaTime;
            //gets the points from time value
        Vector3 currentPoint = PointOnParabola(currentTime);
        Vector3 nextPoint = PointOnParabola(nextTime);

            //Line Renderer pointing in the right direction
        transform.LookAt(nextPoint);

        RaycastHit hit;
            //This is where all of the collision happens
            //This should be a switch statement for efficiency, however I havent changed it yet
        if (RayBetweenPoints(currentPoint, nextPoint, out hit))
        {
            if (!hit.collider.CompareTag("Round"))
            {
                if (hit.collider.CompareTag("Ground"))
                {
                    GameObject g = Instantiate(concreteHit, hit.point, Quaternion.Euler(nextPoint - currentPoint));
                    g.transform.parent = null;
                }

                if (hit.collider.CompareTag("Enemy"))
                {
                    Instantiate(bloodHit, hit.point, Quaternion.Euler(nextPoint - currentPoint));
                    hit.collider.gameObject.GetComponent<Enemy>().Hit(Random.Range(damage.x, damage.y), hit.point);
                }
                if (hit.collider.CompareTag("EnemyHead"))
                {
                    hit.collider.gameObject.GetComponent<Head>().Hit();
                }
```

```
            if (hit.collider.CompareTag("Player"))
            {
                Transform player = GameManager.GM.player.transform;
                Vector3 hitPoint = player.position;
                hitPoint.y += 1.75f;
                hitPoint += 0.5f * player.forward;
                Instantiate(bloodHit, hitPoint,
Quaternion.Euler(nextPoint - currentPoint));

                hit.collider.gameObject.GetComponent<Player>
().Hit(Random.Range(damage.x, damage.y));
            }
            if (hit.collider.CompareTag("Target"))
            {
                AudioSource g =
hit.collider.gameObject.GetComponent<AudioSource>();
                g.PlayOneShot(g.clip);
            }
            Destroy(gameObject);

        }
    }
```

This solution for bullet physics works perfectly, and as it is raycast-based, there will be no issues with hit detection as raycasts are much more consistent. The only issue I have is that the bullet drop cannot be tuned to real-world values. This is because the calculation is `gravity * time * time,` and `gravity` is not based on any real world physics.

UI and Infiltration - Mar 21st - Apr 22

Now that I had the player character functional, I decided to create the Infiltration screen, which is where you choose your loadout and the point that you want to enter the map. I used a camera that is rendering the map from above, and some simple buttons on the side of the screen. Right now, the UI is quite unintuitive and many people glazed their eyes over the buttons to select the infiltration point while testing. I think this is because I need larger icons that will contrast more with the background, as well as some text labels. I could additionally grey out the Infiltration button until both a weapon and an infiltration point is selected.

As well as this, I added some UI to the game including a crosshair, health indicator, ammo indicator and fire mode indicator. The crosshair uses a raycast directly from the barrel and displays the crosshair wherever it hits. This means that when you are moving around, the crosshair will accurately reflect the point where the bullet will hit, and it will move around with weapon sway as well.

AI - Apr 11th - Apr 26th

AI was the feature that I expected would be one of the hardest features to implement, however once I started it was more intuitive than I thought. Unity's Navmesh system is very high quality, and I was able to bake a navmesh (a mesh with every area that an enemy is able to walk to) with minimal effort. Then I am able to give the enemy a point, and it will automatically pathfind to that location with smooth movement. In order to

implement the Finite State Machine, I created an enum, which is essentially a global variable type that pairs an integer up with a string.

```
public enum EnemyState
{
    Idle,
    Attack,
    Move
}```
<span style="color:#FF0000">
Then, I needed to create the methods that hold the rest of the code. Each state has 3 methods: Enter, Update and Exit. In the FixedUpdate(), there is a switch statement that changes which Update method gets called depending on the current state.
</span>
```cs
switch (state)
        {
            case EnemyState.Idle: UpdateIdle(); break;
            case EnemyState.Attack: UpdateAttack(); break;
            case EnemyState.Move: UpdateMove(); break;
        }```
<span style="color:#FF0000">
The Enter and Exit methods are fairly simple and now we have 3 states that we can switch between and a seperate Update method for each state.
</span>
```cs
    void EnterMove()
    {
        state = EnemyState.Move;
        Vector3 pos = transform.position;
        pos.x += Random.Range(0, 10);
        pos.z += Random.Range(0, 10);
        agent.SetDestination(pos);
    }
```

```
    void ExitAttack(EnemyState nextState)
    {
        switch (nextState)
        {
            case EnemyState.Idle: EnterIdle(); break;
            case EnemyState.Move: EnterMove(); break;
        }
    }
```

To begin with, the enemies are all in the Idle state, in which they will look towards the last position that the player was seen (or in the default rotation if they have never seen the

player). Before I wrote the code to tell the enemies where to go, I decided they should be able to shoot back. To do this, I needed to check if the enemy could see the player, which was as simple as measuring the angle of the raycast between the enemy and the player, and if it was inside the "view cone" bounds, then the enemy could see you.

```
if (Physics.Raycast(eyePos.position,
GameManager.GM.player.transform.position - eyePos.position, out
RaycastHit hit, Mathf.Infinity))
{
    if (hit.transform.CompareTag("Player") &&
Vector3.Angle(lookPos.forward, GameManager.GM.player.transform.position
- lookPos.position) < viewCone)
    {
        canSeePlayer = true;
        lastPositionPlayerSpotted =
GameManager.GM.player.transform.position;
        timeSincePlayerSeen += Time.fixedDeltaTime;
    }
    else
    {
        canSeePlayer = false;
        timeSincePlayerSeen = 0;
    }
}
```

Once that check was in place, the enemy will now transition into its `Attack` State if it spots you, then after a randomised "reaction time" (0.2s - 2s depending on distance from player), they will start firing in bursts at you. If they are a long distance away, they will fire in bursts of 1 or 2 shots, however if they are closer they may use full-auto fire. This was because I noticed enemies very far away would be able to shoot extremely accurate full-auto bursts at an unrealistically long range. There is also a short, randomised cooldown between burst shots. Now that the enemies can shoot back in a realistic manner depending on the distance from the player, I added a simple "whizz" mechanic where the enemy can detect if a bullet has barely missed them, and turn in the direction of where the shot came from. Then it was time to add the movement to the AI enemies. This was one of the hardest parts, as I used weighted chance to decide what the AI should do. Between each burst of shots, it will decide whether or not to change its state to Move. This means that they may initially stand in place and shoot back, but then reposition after certain variables change. The method for calculating weighted chance takes into account the current health, if any teammates are nearby, the current number of rounds in the enemies mag, and a variable called "`surprise`." `surprise` is the angle between the enemy's look direction and the direction toward the player. This means that if you fire a shot at the enemy when they are looking the opposite direction, they are more likely to transition to the Move state and run away from you instead of staying put.

```
void UpdateAttack()
{
    if (canSeePlayer)
    {
        firearmPos.LookAt(lastPositionPlayerSpotted);
        firearmPos.localEulerAngles += Random.Range(-0.5f, 0.5f) *
transform.up;
        firearmPos.localEulerAngles += Random.Range(-0.5f, 0.5f) *
transform.right;

        if (timeSincePlayerSeen == 0)
        {
            curReactionTime = Random.Range(reactionTime.x,
reactionTime.y);
            onCooldown = true;
            CooldownShot();
            //roundsLeftInBurst = 1;
            ResetShot();
        }
        else if (!onCooldown && roundsLeftInBurst ≤ 0)
        {
            onCooldown = true;
            Invoke(nameof(CooldownShot), Random.Range(cooldown.x,
cooldown.y));
        }
        if (roundsInMag ≤ 0)
        {
            isReloading = true;
            Invoke(nameof(Reload), reloadTime);
        }

        if (!isReloading && !onCooldown && canShoot && roundsLeftInBurst
> 0 && timeSincePlayerSeen > curReactionTime)
        {
            Instantiate(bulletPrefab, firearmPos.position,
firearmPos.rotation);
            canShoot = false;
            roundsLeftInBurst -= 1;
            roundsInMag -= 1;

            Invoke(nameof(ResetShot), 0.1f);

            anim.Play("Base Layer.demo_combat_shoot");
            source.PlayOneShot(shotSounds[Random.Range(0,
shotSounds.Count)]);
        }
    }
}
 void ResetShot() ⇒ canShoot = true;
```

```
 void CooldownShot()
 {
     float healthChance = Mathf.InverseLerp(0, 100, health);
     float teamChance = Mathf.InverseLerp(0, 4, nearbyTeam.Count);
     float magChance = Mathf.InverseLerp(0, 10, roundsInMag);
     float surpriseChance = Mathf.InverseLerp(180, 0, surprise);

     float chanceToMove = Mathf.InverseLerp(0, 3, healthChance +
magChance + surpriseChance + teamChance);

     float decision = Random.Range(0f, 1f);

     ExitAttack(EnemyState.Move);

     float maxShots = Mathf.Lerp(10, 1, distFromPlayer);
     roundsLeftInBurst = (int)Random.Range(1, maxShots);
     onCooldown = false;
 }
```

This makes the AI feel much more reactive and like they are actually thinking, which helps immersion greatly. If you catch the enemy by surprise, they will be more scared and run away, but if they are expecting you and have many friends nearby, they may be more aggressive and push you. I am very happy with how this AI has come out and it feels like a thinking agent that can execute different tactics very well. I added blood effects to the enemy, so it was obvious when you hit them, as well as a separate head hitbox that would instantly kill them if you hit a headshot. Finally, I also added ragdolls when they die, so they fall to the ground in a realistic manner. I also added a debug menu that shows current state, health, if they can see the player and magazine count.

After gathering some feedback, I decided that a sprint feature would be very useful, and used some of my resubmission time to implement this. It was fairly simple, however I

needed to edit the FirearmInfo class in order to add 2 new Vector3's: sprintPos and sprintRot, as well as adding a stamina stat. I also added a freelook feature, allowing you to look to the side as you sprint or walk around without disrupting your movement.

```
if (inputSprint && stamina > 0 && Input.GetKey(KeyCode.W))
{
    isSprinting = true;
    swayController.isAiming = false;
    isWalking = false;

    maxSpeed = sprintSpeed;
}
else
{
    isSprinting = false;
    inputSprint = false;
}
```

```
if (inputFreelook > 0.5f)
{
    freelookAngles += inputView * 5;
    freelookAngles.x=Mathf.Clamp(freelookAngles.x, -65, 65);
    freelookAngles.y=Mathf.Clamp(freelookAngles.y, -65, 65);

    GameManager.GM.playCameras[0].transform.localRotation =
Quaternion.Euler(-freelookAngles.y, freelookAngles.x, 0);

}
else
{
    cameraAngles += inputView;
    freelookAngles = Vector2.zero;

    cameraAngles.y = Mathf.Clamp(cameraAngles.y, -90, 90);

    Quaternion camRot = Quaternion.AngleAxis(-cameraAngles.y,
Vector3.right);
    Quaternion playerRot = Quaternion.AngleAxis(cameraAngles.x,
Vector3.up);

    camHolder.localRotation = camRot * Quaternion.Euler(camRecoil);
    transform.localRotation = playerRot;

    GameManager.GM.playCameras[0].transform.localRotation =
Quaternion.identity;
    GameManager.GM.playCameras[1].transform.localRotation =
```

```
Quaternion.identity;
}
```

Testing and Feedback

Testing and Feedback

Student 1

The game in its current state is playable, which at the very least meets the client brief. On top of that though the game has a lot of depth, with a variety of weapons, spawn locations and assets found within the map. Each gun has its own recoil, magazine size and overall feel. The game is definitely replayable and I could see myself going back to try out different builds and playstyles.

Some areas for improvement would be the games feedback. Visual and audible feedback would help elevate the game ten fold. I think this game would benefit from a damage indicators, gun shot audio and a medical symbol near the HP numbers. This kind of feedback would make the game more responsive and I would feel less cheated out of death. A simple Health logo PNG would suffice and for the other suggestions I'm sure you could look to YouTube or begin experimenting. As well as this, there are some areas that you can't shoot through, such as the wire fences and an antenna at the military base

Student 2

- the pro of the game is that it feels hyper realistic.
- A con is that some players might find it difficult if they are beginners.

Student 3

- Could add some information about what each gun does, currently there are just names with no explanation.
- The jumping is highly inconsistent.
- The map is set up in such a way that there is no reason to pick a weapon other than a sniper rifle.
- There is currently no way to exit the game.

Student 4

- The shooting mechanics are very accurate.
- The map diagram is very clear and shows the different areas of the map well.
- Players should be able to shoot through the barbed wire walls.

- The jumping is quite overpowered and makes the player travel too fast.

Jacob

- Reloading whilst scoped keeps you locked into 'zoomed in' movement- limits movement speed and aim- is this intentional? Jumping sometimes works, standing still jump is reasonable height (sometimes you have a baby jump). Run and jump is too high- slides to far, is too fast and '#unrealistic' for a simulation/tactical shooter.
- Game would benefit from crouch and possibly prone. Also an indicator as to where you are being shot from or audio.
- However, overall, the gunplay is excellent, recoil feels reasonable for the gun (DT MDR). A.I is mainly responsive (increase turn speed and reaction time).

Student 6

- Indicator of where you are getting shot from around crosshair
- Jumping is broken

Tommi

- correct names showing in menu
- jumping is not working all the time
- movement speed is slow maybe a speed boost or a closer spawn
- audio would be nice for knowing directions of enemies
- some textures are broken on vehicles Normals are facing wrong direction

Friend

- add max distance for enemy spotting you
- infil screen needs a lot of work
- add more cover on the map
- add sprint key
- ragdolls should have no collision
- ADS on press not release
- give weapon crosshair a black outline

Response

I agree with a lot of the feedback on the character movement that was given, especially the jumping issue. This is a fairly simple fix that I implemented around 1/4 of a day. It broke the game in many ways and promoted an extremely unrealistic playstyle where you would jump around the map at extreme speeds. I also agree with the need for

crouch and prone positions and possibly a sprint feature as these were originally planned but I ran out of time to add them.

Another large issue was that many people didn't know where they were being shot from, which led to a bit of frustration as they were seemingly getting killed from nowhere. I think the biggest reason why this happened was a lack of audio feedback. I added gunshot sounds to the player and enemy that uses Unity's directional audio. This means that you can better place the direction that the enemy is shooting from as you can hear their gun fire. Another person suggested that there should be a damage indicator around your crosshair (similar to Halo or Fortnite). While this would fix the issue with players not knowing the direction they are being shot from, I don't think I will add this feature as it is quite unrealistic and does not fit the minimal style of the HUD that I have decided on. Adding tracers to the enemy bullets and audio to the shots seems to be enough to place where the shots are coming from.

There were also some issues with the map design, which I think exacerbated the issues with players detecting where they were being shot from. The map that I designed, Clifftown, is very open and has many people stationed on roofs and in sniper towers, meaning that many enemies are >100m away and you need the DT MDR with an 8x scope in order to have a fighting chance. Using guns intended for Close-Quarters Combat (CQB) was not a viable tactic at all. If I were to make a second map, I would add much more cover on the ground level and less sniper enemies.

There are also some smaller Quality of Life issues that were brought up, such as the weapon staying "scoped in" while the reload animation was playing, meaning that your sensitivity, movement speed and FOV stayed very low while reloading instead of un-scoping. There were also a few props such as wire fences that you were not able to shoot through even though you should be able to.

Evaluation

Evaluation

The feedback I got was extremely useful in balancing different aspects of my game, and even though I was not able to get every feature I planned in, I think the game has come out well and is quite fun to play. It has a functional AI system that can handle over 40 enemies in a single map and a smooth feeling character controller with realistic bullet physics and flawless hit detection. I am particularly proud of the weapon sway and movement, including procedural breathing and walking, as well as the enemy AI that feels like it is making tactical decisions.

I think I may have been a bit too ambitious, especially with the AI system. I originally wanted to have a friendly teammate as well as the enemies, however by the time I was finished with the enemy AI, the hand-in date was too close. It required additional logic for

the UI, as well as a more complex system that can take orders from the player (to suppress an area, cover your back, etc). My research into the Impact of AI was extremely useful in creating my own AI, as it taught me many different methods for creating an AI system.

Additional movement options such as sprinting, crouching and going prone would be good to add as well, as it would improve the stealth mechanics and add the ability to snipe people with less weapon sway. I think the game loop was well thought out and I didn't have to modify the game mode at all. In my GDD, I explained that there would be different objectives around the map, and I think if I had more time then I would definitely add some simple objectives such as needing to pick up a laptop of intelligence, rescue a prisoner or clear a specific area of enemies.

During the resubmission, I added sprinting, freelook and refined the AI with a Finite State Machine, as well as some post-processing effects on magnified optics. I think this was a good use of my time, as it deepened my understanding of Finite State Machines and other AI systems, including the upsides and downsides of different solutions.

Resub

All Merits

- Research into other games (tarkov?)
- Is difficulty balanced?

Good uses of your English literacy throughout your documentation. It would be nice for you to further explain your reasons for why you decided to work. Make sure you are justifying your process.

technical skills are really coming along as you progress through the ports. It is nice to see how much you have developed throughout.

With your presentation. You did a really good job at presenting to me and the others. It was nice to see that you were talking to others to get feedback. Your communication skills are really coming along. However, it be nice to see expansions on your justifications. Continue to work on your communication, confidence, and creativity skills as these will really help you in your future. I would recommend that you talk to your team during the group project. Continue to write down your process. And write down any idea that comes to your mind, No matter how small.

There is an effective number of informing ideas in your documentation. It would be good to have more links throughout and make sure there is a wide range of research techniques used.

Your problem-solving skills are quite clear and sometimes effective, however. There needs to be more ideations and showing more of your concept skills.

You need to link in your Research into your technical skills to achieve the higher criteria. There is some clear evidence of your technical ability here, however. Make sure you are documenting your entire process throughout.

There is an effective amount of professional practice in your documentation and in your pitch, ensure that you are always updating your time plan and are testing your animation with others.

You have some clear communication throughout; however, your documentation needs further work in the form of your page layout and structure. I would encourage you to review your testing results and go back over your work.