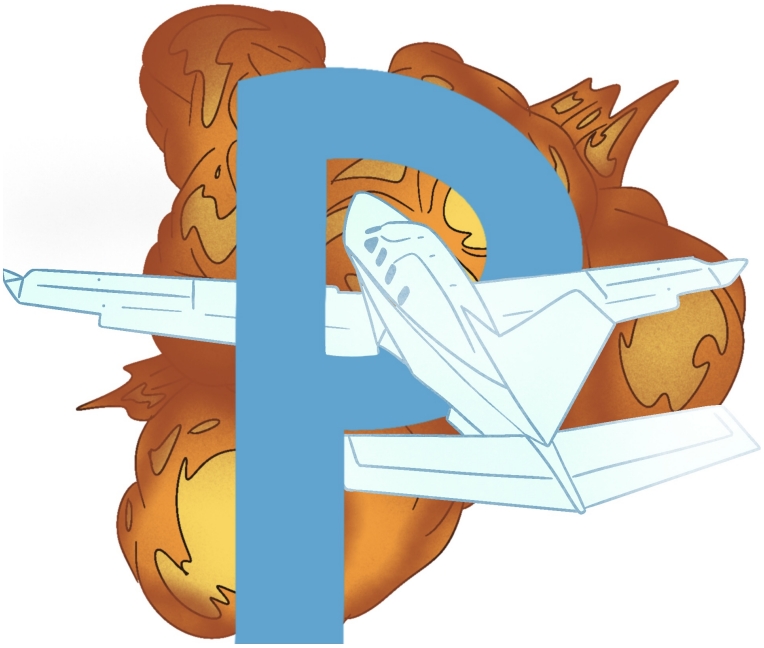
Achille Guerard
Térence miralves
Timothy Pearson
Léopold Tran

Project Pegasus

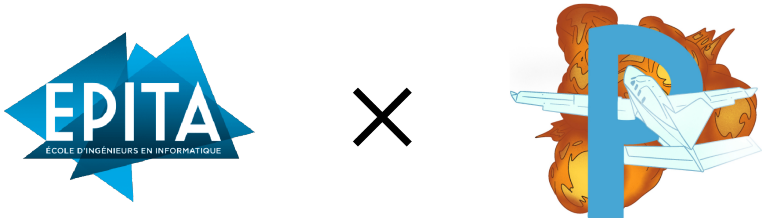
SECOND REPORT



EAF

prépa sup

25th April 2022



Project Pegasus

Contents

1 Introduction

This report will serve as a follow-up to the book of specifications and the first report, as well as a summary of the tasks completed since the previous report's distribution. The difficulties that were faced and the solutions that were adopted will be explored.

As a quick reminder, Project Pegasus is a multiplayer air combat game which includes a multiplayer mode as well as a single player mode. We, the EAF team, aim to make this game as realistic as possible through physics and design. Project Pegasus will also feature an AI that will control plane movements.

Since the first defense, we have worked on numerous aspects of the game. Not only have we continued to produce planes and maps but we have also developed a whole new way to interact with the game thanks to the development of our very own AI.

Indeed, our AI is capable of following a target, getting into an attack position and then choosing the appropriate gun according to the AOA, distance, and availability of the weapons. We plan to implement energy managment of the plane according to physics law in order for the AI to outsmart the player.

We have also implemented a HUD in order to improve the players gaming experience while conserving the realism of the flight simulator.

2 General progression

2.1 AI

Since last defense we have built an AI that is capable of following a target, getting into attack mode, and shooting the target while avoiding the ground. Indeed the AI's primary objective is to avoid the ground, then it is to follow the enemy plane and then it is to shoot it down. In order to flight the AI uses the same components as the player's plane with the addition of an AI controller class.

The AI calculates the distance, AOA, and position it needs to reach in order to maximize its chances of success. When it has reached this position it chooses between the available weapons and tries to shoot down the target.

2.2 Weapon system

The weapon system we have developped is composed of two different weapons available according to the plane. For example the propeller plane only has cannons while the jetfighter has a cannon and missiles.

The cannon is fairly straighforward, it requires for the instantiated bullet to be synced between all occurences of the game just like it was explained during the first report.

The missiles also need to be instantiated accross all sessions. On top of a simple missile launcher we have developped a missile locker, which means that the missile is able to, once locked on to a target, follow the plane in order to shoot it down. To keep the game interresting and realist we have implemented the contraint that the missile cannot follow if the plane does "evasive" manueuvers, the missile locker is simply a way to make sure the missile reaches its target if the latter does move too much.

2.3 Level design

In the beginning, we didn't really put a lot of effort into map making, as we only used the map for testing purposes. But as we came to a point where we needed some definitive maps, we really had to master the art of map making. At first, we didn't really know how to start, so we did some research on the internet to learn how to make maps. It is a rather long and painful process. The first thing we did was choose every

element we wanted and its scale. Then, we added the terrain, which at first appears plain. So, we shaped it with various brushes. After being done with the geography of the map, we implemented the textures. We applied the texture to the ground with brushes. An important point we noticed was that it is vital to use more than one texture for every element of the map in order to break the monotony of the first texture and to make it more realistic. Finally, we made our map real by spawning trees randomly on the terrain. To sum up, we went from making maps for testing purposes to having 2 beautiful and fairly big maps of eighty one kilometers square each.

2.4 Particles effect

The first thing we did with the particles effect was a little trail on one of our prototype planes. It was really basic and wasn't that good, but still, the job was done. Further ahead, we had to add some smoke particles that would come out of burning tanks in the desert map. For this, we had to be creative because, in the beginning, the particles looked like fireflies. At first sight, it was hard to transform them into smoke. But, by learning how to change the shape of emission of the particles, their numbers, or even their materials, we managed to create some convincing smoke. Then, we did the afterburner the jet fighter plane. For this, we learned how to use a script on particles in order to increase the particle effect if the player was stepping on the gas pedal. The last thing we did was the trails that you can see on the wing of the jet fighter. We had to learn how to use trails, and, more precisely, the ribbon trails to play around with their sizes and numbers in order to make a continuous trail that would make the flying natural and smooth for the player. In conclusion, we went from simply changing the number and speed of the particles as well as their material to modifying them with a script and finally using trails.

2.5 UI

To start with, we discussed some basic themes that would complement the rest of the game and decided to go with a futuristic, transparent blue concept so that the game scene would be visible through the canvas. After the first panel with buttons was created, it was very easy to create the others as we simply duplicated it and changed the text. Initially, we tried using a switch statement which opens the corresponding scenes using a tag. However, this method uses the fixed update function which is actually unnecessary and costly in terms of complexity. The solution was to use on click events which call the corresponding functions in the ButtonHandler.cs script which changes the scene using the index of scenes in the build settings. In terms of navigation, when the game first starts, a main menu scene opens which just contains a canvas. When either one of the buttons is pressed (Multiplayer or Single player), the right scene opens and the game starts. When the P key is pressed another menu comes up with two buttons; one for the controls manual and the other to go back to the previous menu.

2.6 HUD

Crosshair aim, health bar and offscreen enemy/missile indicator were the HUD features we wanted to implement. The easiest out of the three was the crosshair which simply consists of two ui images arranged in a triangle shape. Since the camera is fixed for the internal pilot's view, the cross hair doesn't require any movement; however for the external view, the angle between the direction of the camera and the bullets projectiles are too significant for a crosshair to be useful. Therefore, at the moment, it is only implemented on the internal view. Next, the health bar was relatively quick to set up. It consists of two ui images: the overlap and the RectTranform property of the bar set to middle stretch so that it can shorten itself only on the right side. A health variable used in another script allows us to vary the length of the health bar. Finally, the off screen indicator was the most difficult out of the three. In the beginning we tried to install a free unity asset and apply it to our scene. We found the manual quite vague and could not get it working on our own canvas. So we decided to write our own. This started with using ray casts from the centre of the canvas to the x,y position of the target on the canva plane. Since the canvas had to be in Screen Space - Overlay mode, its position does not change relative to the plane. Then we used the plane a reference, created a edge collider around the board of the canvas, found the vector direction to the target projected on the plane normal to the forward direction of the aircraft (local z axis) and then cast a ray from the centre of the canvas to in the correct direction and then use a RaycastHit to find the collision point to set the

indicators position to. This became quite confusing and difficult to debug so we thought of an alternative and much simpler method. The local direction vector normalised multiplied by a constant which is added to the centre point of the canvas would give the position for the indicator around a circumference. After that, we changed the image to an arrow and added a line in the script that rotates it so that it points in the right direction using the inverse tan of the y component divided by the x component. Finally, when the target is in the camera's frame, we want the arrow to be disabled and instead a red translucent square to hover over the target. This took some time however was made very easy thanks to the WorldToScreenPoint function provided by unity.

2.7 Creative production

During this second period, we worked a lot on making our game beautiful, both in appearance and in play. We also made a point of making our game even more realistic than it was before. The thing that best embodies our search for realism is our third and last plane: the jet fighter. This plane was modelled on real plans and textured in a style that matches reality. Moreover, it has been fitted with missiles and greatly improved particle effects, which makes the whole thing almost bewildering. Another big part of this creative process was adding sound to a muted game. Indeed, two or more planes fighting without a single sound is quite disturbing. That is why we chose to implement the sound and music early. This improvement makes a great difference in the way the player can enjoy the game.

Task	Prevision	Actual progress
Multiplayer	80%	80%
AI	60%	70%
Gameplay	80%	60%
3D modelling	750%	90%
2D modelling	75%	90%
3D animation	20%	75%
Physics	90%	90%
Particles effect	80%	70%
UI	60%	60%
Level design	60%	70%
Tutorial	50%	55%
Sound effect	20%	70%
Music	20%	100%
Website	90%	90%

3 Leopold's progression

3.1 AI

The artificial intelligence will flight the plane and follow the target while avoiding the ground, it will then get into an attack position and when ready will attack its target.
When feasible, the AI will strive to avoid being attacked, dodge missiles, and use its weaponry.

The player must combat one AI jet that is flying about. To observe an AI against AI combat, the user can also enable AI on their own aircraft.

3.1.1 Navigation

3.1.1.1 Steering

The AI's primary objective is to aim its nose towards the target. This one behavior is responsible for the majority of the dogfight's atmosphere. When the player is in front of the AI, the AI turns repeatedly to get behind the player, which has the unintended consequence of evading the player's assaults.

It's not difficult to figure out which way to go. Because the plane is most maneuverable when pitching up, the AI seeks to align itself by rolling in the direction of the goal.

```
var error = targetPosition - plane.Rigidbody.position;
error = Quaternion.Inverse(plane.Rigidbody.rotation) * error;   //transform into local
    space
```

The position of the two planes in world space doesn't matter. Steering is calculated based on the position of the target plane in local space.

```
var pitchError = new Vector3(0, error.y, error.z).normalized;
var pitch = Vector3.SignedAngle(Vector3.forward, pitchError, Vector3.right);
if (-pitch < pitchUpThreshold) pitch += 360f;
targetInput.x = pitch;
```

To get pitchError, the error value is flattened into the YZ plane. Vector3 receives this information. To determine the angle needed to pitch up or down to aim at the target, use SignedAngle. The aircraft receives a pitch input of 0 when the target is immediately in front of it.

If the angle is a significant pitch down, the move is transformed to a pitch up. If pitchUpThreshold is set to 15, the AI will always try to pitch up when the target plane is greater than 15 degrees below the AI plane. It will pitch up or down naturally if the angle is less than the threshold. This prevents the plane from trying to pitch down by a substantial amount.

```
var rollError = new Vector3(error.x, error.y, 0).normalized;

if (Vector3.Angle(Vector3.forward, errorDir) < fineSteeringAngle) {
    targetInput.y = error.x;
} else {
    var roll = Vector3.SignedAngle(Vector3.up, rollError, Vector3.forward);
    targetInput.z = roll * rollFactor;
}
```

The XY plane error is used to determine rollError. This rollError, on the other hand, is utilized to compute either the roll or yaw inputs. If the AI plane is pointed at the target at a tiny angle, fineSteeringAngle, it will aim with the rudders for more precise control. The AI will now be able to aim its gun.

The roll error will force the plane to roll, allowing a significant pitch up maneuver if the AI is larger than fineSteeringAngle. When the target is immediately overhead, the roll error is 0.

```
targetInput.x = Mathf.Clamp(targetInput.x, -1, 1);
targetInput.y = Mathf.Clamp(targetInput.y, -1, 1);
targetInput.z = Mathf.Clamp(targetInput.z, -1, 1);

var input = Vector3.MoveTowards(lastInput, targetInput, steeringSpeed * dt);
lastInput = input;
```

Finally, on each axis, the targetInput is constrained to avoid anything like an input of (360, 0, 0). To add a tiny delay to the inputs given by the steering function, we utilize Vector3.MoveTowards.

3.1.1.2 Ground avoidance

The steering logic described above allows the AI to navigate toward a destination, but additional reasoning is required to prevent the AI from crashing. While chasing its prey, the AI will gladly fly into the earth.

By projecting a beam 10 degrees below where its nose is aiming, the AI avoids colliding with the earth. If the ray collides with the ground, a ground avoidance algorithm takes control and steers the plane away

from it.

The plane will level off, pull up as hard as it can, then slow down if required until the ground detecting raycast no longer contacts an obstruction.

```
Vector3 AvoidGround() {
    //roll level and pull up
    var roll = plane.Rigidbody.rotation.eulerAngles.z;
    if (roll > 180f) roll -= 360f;
    return new Vector3(-1, 0, Mathf.Clamp(-roll * rollFactor, -1, 1));
}
```

This method returns the necessary steering input to prevent hitting the ground. The rollFactor parameter is a modest integer, such as 0.01, that is used to decrease oscillation when rolling. On the X axis, the steering vector is set to -1, the greatest pitch up.

3.1.1.3 Getting into attack position

3.1.2 Attack

To fire the cannon, the AI must first arrive within range of the weapon and direct the plane's nose at the predicted lead point. If it is out of range, it will just point its nose at the target instead of firing.

```
Vector3 GetTargetPosition() {
    var targetPosition = plane.Target.Position;

    if (Vector3.Distance(targetPosition, plane.Rigidbody.position) < cannonRange) {
        return Utilities.FirstOrderIntercept(plane.Rigidbody.position, plane.Rigidbody.
            velocity, bulletSpeed, targetPosition, plane.Target.Velocity);
    }

    return targetPosition;
}
```

Once in range, the AI waits for the nose to be within a minor angular error threshold before firing the gun.

```
var targetPosition = Utilities.FirstOrderIntercept(plane.Rigidbody.position, plane.
    Rigidbody.velocity, bulletSpeed, plane.Target.Position, plane.Target.Velocity);

var error = targetPosition - plane.Rigidbody.position;
var range = error.magnitude;
var targetDir = error.normalized;
var targetAngle = Vector3.Angle(targetDir, plane.Rigidbody.rotation * Vector3.forward);

if (range < cannonRange && targetAngle < cannonMaxFireAngle && cannonCooldownTimer == 0) {
    //fire cannon
}
```

There is a range check while firing missiles, but there is no need to lead the target plane because the missile does it for you.

```
var error = plane.Target.Position - plane.Rigidbody.position;
var range = error.magnitude;
var targetDir = error.normalized;
var targetAngle = Vector3.Angle(targetDir, plane.Rigidbody.rotation * Vector3.forward);

if (!plane.MissileLocked || !(targetAngle < missileMaxFireAngle || (180f - targetAngle) < missileMaxFireAngle)) {
    //don't fire if not locked or target is too off angle
    //can fire if angle is close to 0 (chasing) or 180 (head on)
    missileDelayTimer = missileLockFiringDelay;
}

if (range < missileMaxRange && range > missileMinRange && missileDelayTimer == 0 && missileCooldownTimer == 0) {
    //fire missile
}
```

3.1.3 Dodging Missiles

Not only must the AI be able to use its weaponry, but it must also be able to avoid incoming strikes. The AI's steering behavior is the only logical way to avoid artillery fire. The AI is continually spinning around to be behind the player, which means it is continuously evading cannon fire.

While there is no specific reasoning for avoiding bullets, evading rockets has its own code route. Long-range missiles have plenty of time to alter their trajectory and intercept their intended target. As a result, unlike with bullets, the AI cannot simply pivot towards the opponent.

There are certainly a variety of techniques for the AI to avoid projectiles. Calculate four "dodge spots" above, below, left, and right of the projectile using the system I chose. The missile's distance from the dodge locations is the same as the AI's distance from the missile. Choose the nearest dodge point to the plane and maneuver towards it.

Because there are several dodge spots, the plane has a timer to prevent it from changing dodge points too frequently. This prevents the plane from fast switching between two opposing dodge locations, which would cause the plane to fly in a straight path.

3.1.4 AI conclusion

All of these systems work together to produce an AI that is very difficult to defeat. With its guns and missiles, the AI will attempt to line up a shot. It will strive to avoid striking the ground and avoid your strikes. It will try to conserve energy, but this will expose it to assault.

The AI will carry out these functions in a priority order. The priorities from highest to lowest are:

- Ground avoidance
- Missile dodging
- Steering (dogfighting)

By outturning the AI, exhausting its energy and forcing it to recharge, or causing it to fly into the ground, you can defeat it. We hope your ground avoidance and shooting skills procedures are better than the ones we wrote because the AI can do the same to you.

3.2 Weapon system

The planes in this game will be armed with guns and missiles.

Cannons and missiles have one thing in common: they must lead their targets since they are physical things going at a fixed pace. The pilot must supply the lead with guns by directing his nose ahead of the target. Missiles compute their own lead and utilize it as guidance.

To compute lead or interception, we use this code from wiki.unity3d.com in both scenarios. Given the speed of the projectile and the speed of the platform that launches it, this method determines the lead required to reach a moving target.

3.2.1 Cannon

The cannon is straightforward. The plane fires a bullet that goes forward and causes damage to the opponent plane. Because bullets are tiny and rapid, they may be able to "tunnel" through an object. That is, they would not notice a collision and would pass straight through. We don't utilize physics colliders at all to prevent this problem. Instead, raycasts are used to detect all collisions.

We can compute the lead required to strike the target based on the location and velocity of the two planes, as well as the bullet's velocity. Players may see this lead as a reticle on the HUD. When the AI uses the cannon, it will aim at this spot.

3.2.2 Missile lock

The plane must first lock on to the target before the missile can track it.

```
1  //default neutral position is forward
2  Vector3 targetDir = Vector3.forward;
3  MissileTracking = false;
4
5  if (Target != null && !Target.Plane.Dead) {
6      var error = target.Position - Rigidbody.position;
7      var errorDir = Quaternion.Inverse(Rigidbody.rotation) * error.normalized; //transform
           into local space
8
9      if (error.magnitude <= lockRange && Vector3.Angle(Vector3.forward, errorDir) <=
           lockAngle) {
10         MissileTracking = true;
11         targetDir = errorDir;
12     }
13 }
```

Based on range and angle, this code decides whether the target plane can be latched on to. The missile cannot latch on to a target that is too far away or out of boresight, and targetDir is set to the default forward position. If it can lock on, targetDir is set to the target's direction.

```
1  //missile lock either rotates towards the target, or towards the neutral position
2  missileLockDirection = Vector3.RotateTowards(missileLockDirection, targetDir, Mathf.
       Deg2Rad * lockSpeed * dt, 0);
```

The missile is then gently walked towards the target with the missile lock direction.

3.2.3 Missile

Because the missile has its own guidance, it doesn't need to be precisely directed once it has a lock. It will compute the amount of time it will take to reach the objective and steer in that direction. Missiles cause damage by exploding, therefore a direct impact isn't required. The missile will do damage if it bursts near enough to the target.

Real missiles, like the planes that fire them, have energy. They must navigate at a minimal air speed and waste energy in the process of maneuvering and drag. Real missiles have a limited amount of fuel and will glide towards the target until it runs out.

However, in order to keep things simple, we will not design our missiles in this manner. Missiles will not be affected by energy or drag. They will constantly go at the same speed. The rocket just explodes when it runs out of fuel.

```
1 var targetPosition = Utilities.FirstOrderIntercept(Rigidbody.position, Vector3.zero, speed
      , target.Position, target.Velocity);
```

The missile determines the amount of time it will take to intercept the target. Because the missile's speed is supplied as the projectile speed parameter, we report the missile's velocity as Vector3.zero. We only send the missile's speed to the function once because it is the projectile. Otherwise, it would compute the trajectory of a projectile shot at X m/s from a moving platform, yielding an effective velocity of 2X.

Because a projectile traveling twice as fast would require less lead, the lead calculation would result in a significant underestimation of the amount of lead required. As a result, the missile behaves like a "tail chaser." Instead of going to where the target will be, it goes to where the target is. To avoid this, we set the velocity to Vector3.zero.

```
1 var error = targetPosition - Rigidbody.position;
2 var targetDir = error.normalized;
3 var currentDir = Rigidbody.rotation * Vector3.forward;
4
5 //if angle to target is too large, explode
6 if (Vector3.Angle(currentDir, targetDir) > trackingAngle) {
7     Explode();
8     return;
9 }
```

The missile is always aware of its location.
It gets the targetDir by subtracting where it is from where it isn't, the targetPosition. In order to intercept the target, it must face this direction.

currentDir is the missile's current direction of travel.
The missile will detonate if the angle between currentDir and targetDir is too large.
This can happen if the target plane manages to avoid the missile, in which case it explodes at a safe distance away. However, if the missile intercepts and passes the target, the angle will swiftly approach 180 degrees, causing the missile to detonate directly close to the target.

The missile will continue to direct itself if the angle is less than the threshold.

```
//calculate turning rate from G Force and speed
float maxTurnRate = (turningGForce * 9.81f) / speed;  //radians / s
var dir = Vector3.RotateTowards(currentDir, targetDir, maxTurnRate * dt, 0);

Rigidbody.rotation = Quaternion.LookRotation(dir);
```

We can specify the turn rate in Gs induced during the turn instead of degrees per second. So, for example, we can construct a missile with an 8G turn.

```
Rigidbody.velocity = Rigidbody.rotation * new Vector3(0, 0, speed);
```

Finally, we set the missile's speed in the direction it is traveling. This implies that the missile will constantly travel at the same rate. It does not lose energy as a result of turning or drag.

3.3 Website

We had already implemented a website for the first defense that helped us keep note of our progression but we improved it and added some new functionalities.

Figure 1: Homepage

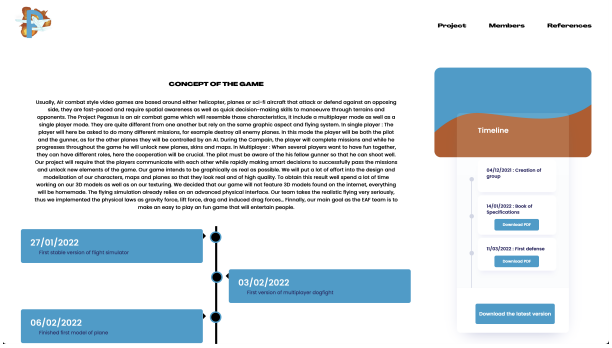


Figure 2: Project page

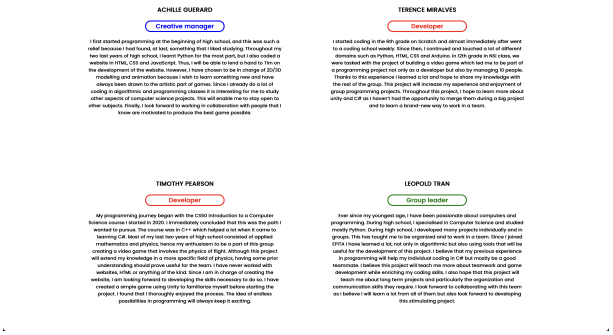


Figure 3: Team page

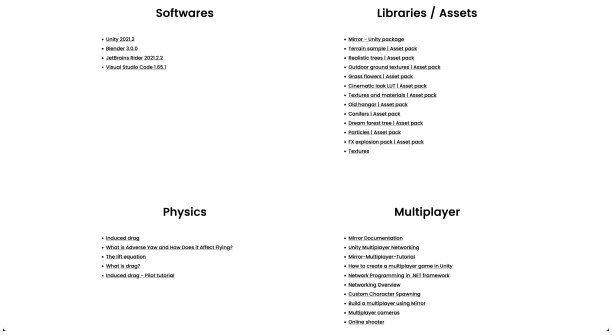


Figure 4: References page

3.3.1 General improvment

The website is a place where people can download our book of specifications, our reports but also the latest version of our game. Our website is also a place where people can learn about our game but also our team.

Since the last defense we have remodeled some aspects in order to improve the navigation and the overall experience of the user. We have also kept the timeline up to date and the download page also.

Because we have added elements such as music and sound effect we made sure to link them in our website with the rest of all the ressources we have and will use during this project.

3.3.2 Embeded version of our game

The major addition to the website is an embded version of our game.

Each team we reach a stable version of our game we push it onto our simmer.io server that is linked to our website.
This enables the users to play the game on the website beforehand. As this is only for the sake of the demo we have choosen not to put all the functionalities of our game on the website such as the multiplayer so that the user downloads the game in order to have the best and full experience.

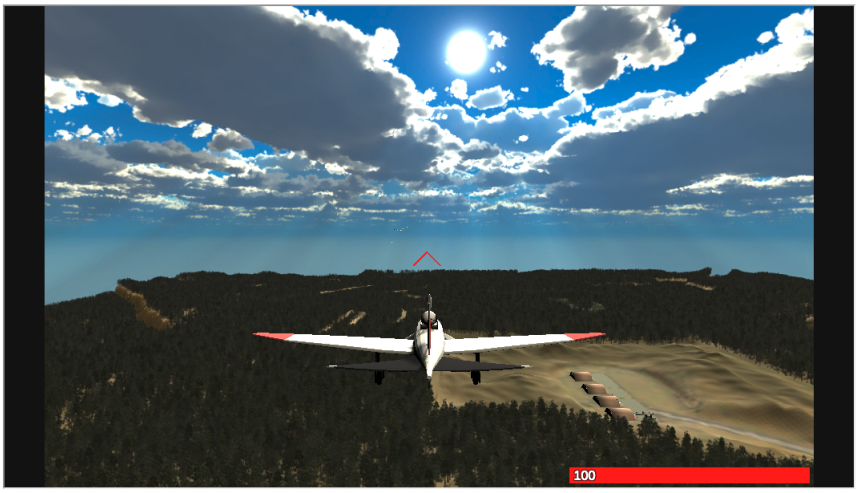


Figure 5: Embedded game

3.4 Music

We had decided that we wanted a song to play throughout our game. Therefore, I choose a song and made an audio listenener that went throughout all the scenes. Then when we jumped from scene to scene we just checked if the music was already playing or not. If it was then we didn't do anything else we started the song.

3.5 Sound interraction

The sound effects include the shot of the cannon, the launch of a missile, a crash and the sound of the engine. For this we choose sounds that fitted the estetic of our game and customized each sound to suit its plane. Then we linked these sounds to an audio source that was played each at the appropriate time.

For bullets, missiles and crash it was fairly simple because we just attached it to the bullet, missile and crash component so when these where instantiated they immited a sound and then were destroyed.

For the sound of the engine it was a little more complicated because we wanted the sound to be proportional to the intensity of the throttle. The fix to this issue was to link the sound to an audio source and then to vary the pitch of this audio source between two points that we had choosen by ear. As the pitch would only vary between -3 and 3 (but actually only between 0.5 and 3 because else it doesn't sound like

an engine anymore) we had to translate our value in order for it to suit.

Once all this was done and we added the sound effect and the music the game took a whole other turn, emerging the player into a more realistic, professional and fun game.

4 Achille's progression

4.1 3D modelling

For this second defence, we thought that we should finish completely the modelling part of this project. In fact, it was very important to us to focus on the other creative aspects of the game. The modelling of planes was a big part of our game, and we are really proud of the result we have achieved. Let us be more specific. Since the last defense, we have added our third and last plane model: the Jet Fighter. This plane is the symbol of our game and will represent our dedication throughout this project. After all, our team is called EAF (Epita's Air Force). As we knew this plane would represent a lot, we discussed and decided during our weekly meeting that it would be featuring missiles (more on that in The part about missiles). Moreover, we have fitted this plane with a yellow and black style that we feel is the best match.

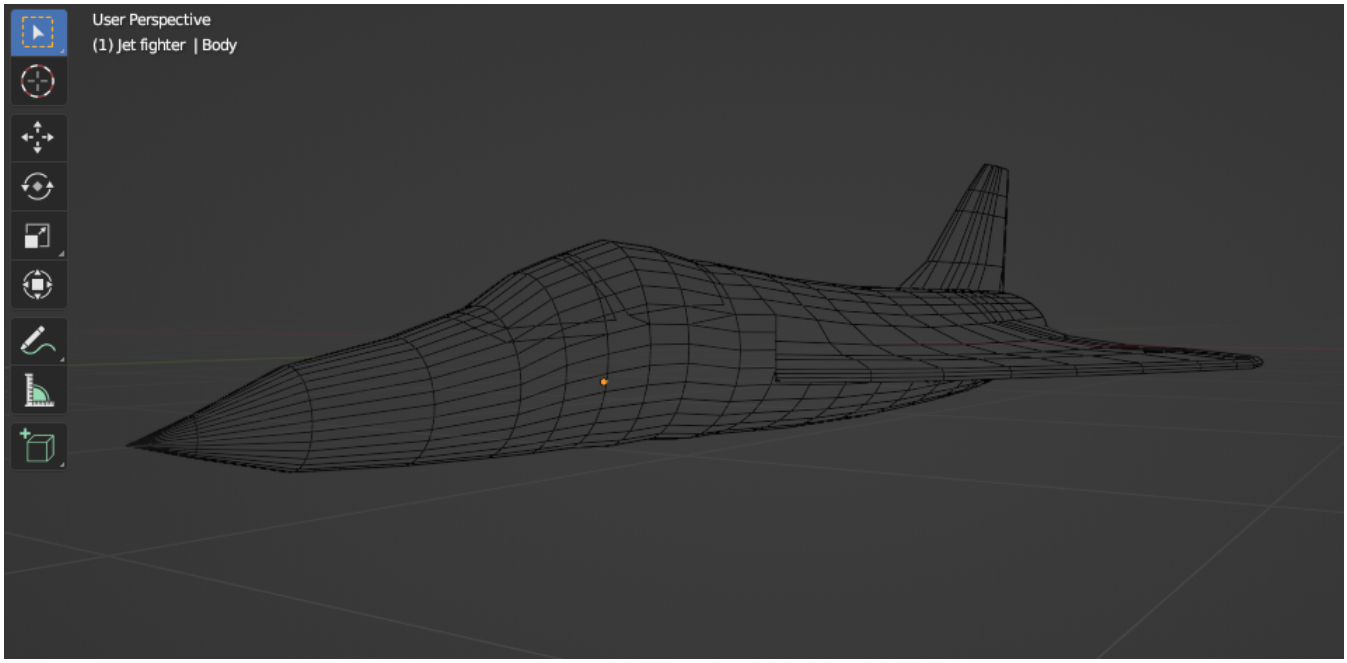


Figure 6: Jetfighter plane without texture

While modeling this plane, a few problems arose:

- The research for style and inspiration.
- Modelling in only one object, without using hierarchy.
- The Seeking of Realism.

These problems were a lot to overcome, but with a bunch of time and effort, we finally succeeded. Here are the solutions we chose. For the style and inspiration, we used a lot of reference images and spent hours on the net looking for some realistic reference images and blueprints of jet-fighters.

This plane needed to look as real as the others, if not more. As a result, we chose to model it as a single object with no parenting (aside from the reactor, which will soon be able to move in sync with acceleration). This choice is due to the fact that a jet-fighter is the most aerodynamic plane you could ever see. No other object should interfere with that. Finally, research was again a big part of the creative process, as you can

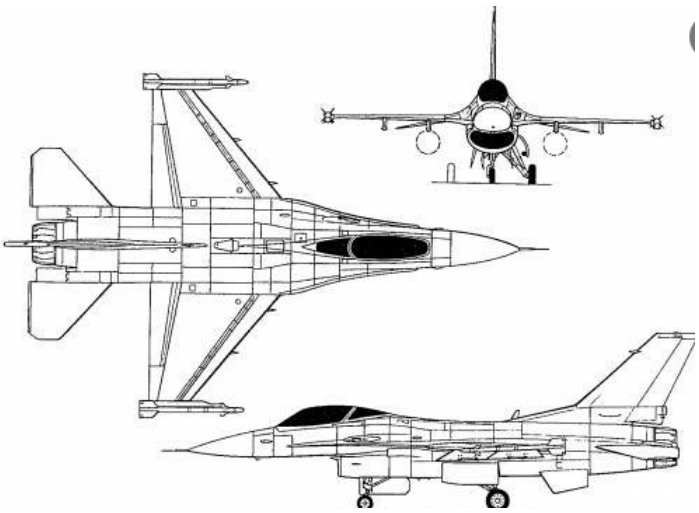


Figure 7: Blueprint of jetfighter

Figure 8: Front view

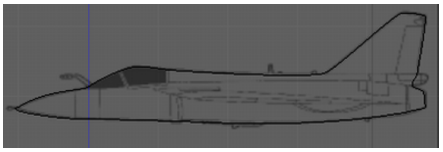


Figure 9: Side view

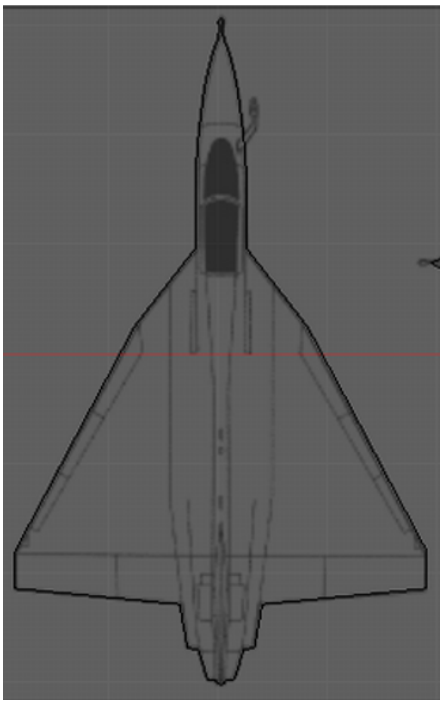


Figure 10: Back view

see from the websites we used in the reference part of this report. The last thing we would like to point out is that the risk of making our own models was worth it. It took some time and it came with loads and loads of struggle, but in the end, we are so proud of the result and we have learned so much about so much. This part of the work required research, creativity, cooperation, and mainly to learn and improve our workflow on different software such as Blender and Unity. This learning was so important for our development as students but also for our game. We now have a game that looks exactly like we wanted it to be because we created it from scratch. Some might choose to do impressive technical stuff and neglect the looks of the game, to us this part is so important that we chose to assign a member of the team only for this.

4.2 Textures

As every model comes with its own texture, we had once again to figure out how to properly "dress" our third and last plane, the Jet-Fighter.

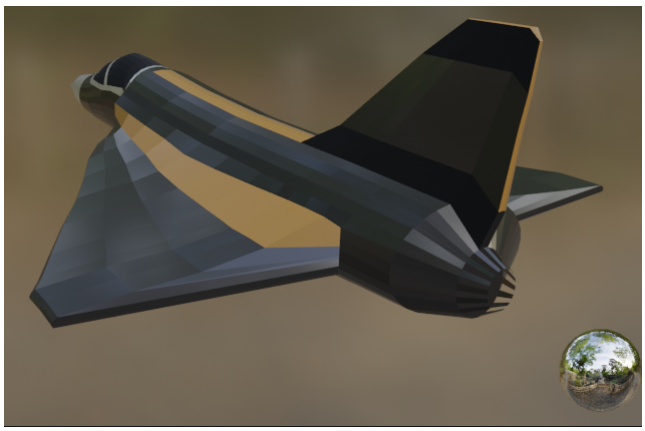


Figure 11: Textured jetfighter

As for every design and creative feature, the responsibility was given to Achille. However, our creative director did not act alone. Every decision is a group decision in our project. That is a rule we set from the beginning and are willing to respect until the end. During the making of texturing, a few problems arose, but no problem did not match its solution. Here are a few examples :

- How to choose the creative aspect of the plane?
- What technique can we use to apply texture?
- How to import and modify textures?

We decided that our jet-fighter needed to embody speed, power, and honour. The texture was built consequently. To embody power, we thought that black was the colour to go with—a metallic black that remains unseen. Then, for speed, we opted for yellow, the colour of lightning. On each side, you can see two straps of yellow paint that evoke the rapidity of our jet fighter. Now that we have discussed the matter of art and the creative features of this texturing, let us be more specific concerning its realisation. We used at first a technique called UV mapping.

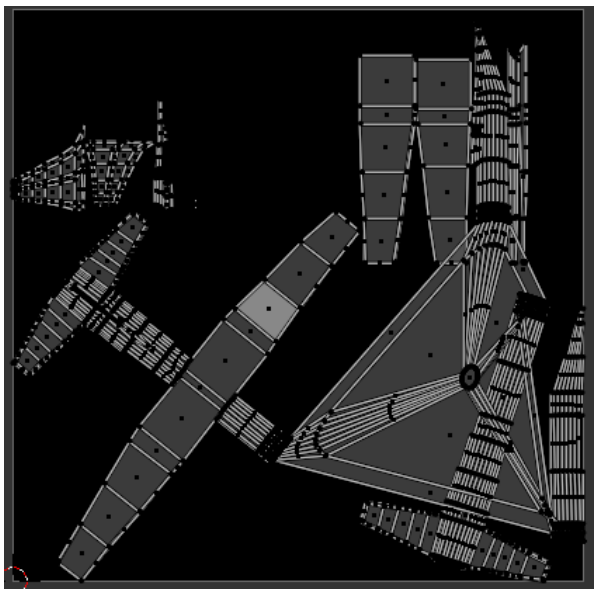


Figure 12: UV mapping of cargo plane

This technique is a workspace of Blender called "UV editing" and we've chosen to use it. UV Mapping consists in converting the 3D faces of our model into 2D on a plane to ease the application of textures. This

technique is very useful when you are dealing with an object that has many different textures and needs you to select some specific faces. As for this jet fighter, we needed to apply 3 different textures to different parts of the plane. The metallic black paint, the yellow one (a little less metallic), and then the glass plane for the cockpit. The use of UV mapping saved us quite a lot of time. Finally, we chose to import textures from the internet as making them is really a job for professionals. However, we decided to modify them so that they would perfectly match our game. To do that, we had to understand the basics of texture making. You can see in figure 27392 "Texture Black Paint" the texture of the metallic black paint and the interface we had to work on.

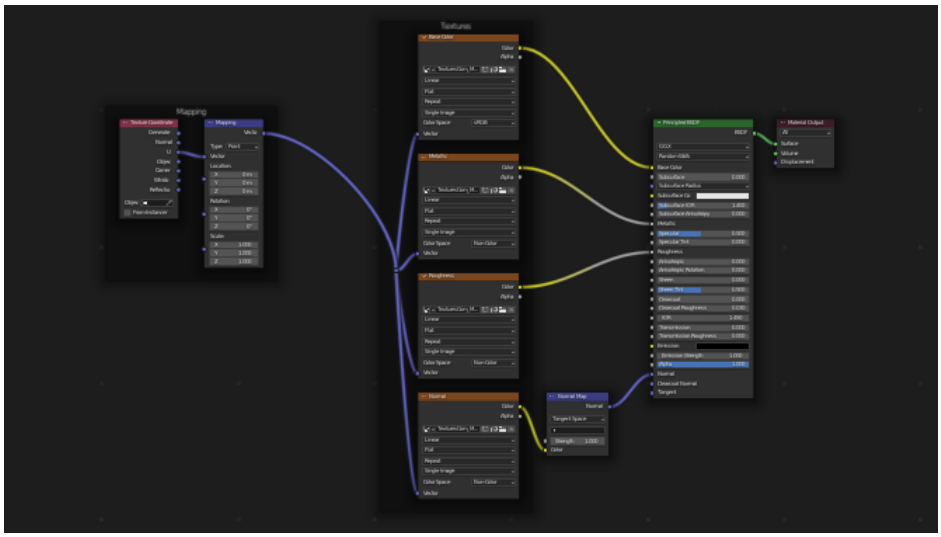


Figure 13: Texture hierarchy of black paint texture

Even though it looks like a bunch of cables, it is simple enough. The boxes are connected from input (left) to output (right). The input here is obviously the mapping we have discussed previously. We checked the boxes "surface" in both the input and output because we only deal with 2D texture in our project. In the middle, you can see all the texture aspects that apply. Every parameter is taken into account: the base colour, the metallic aspect, the roughness, and so on. Finally, you can see on the rightmost side the output on the material surfaces.

4.3 Sound effect

During the development of any type of game, the question of sound effects and music arises. As a team, we thought that our game needed sound effects and music to match the already great design and realism. We encountered a few problems during the implementation of the sound effects and music, for example :

- Finding good sounds libraries to keep a realistic game
- Finding a way to loop our sound correctly

That is why we decided to use recorded sounds from different royalty-free online libraries. This choice matches perfectly the realism we want our game to have. Furthermore, each sound has been mixed and modified inside the software audacity so that it could be pleasant to hear. The mixing was quite a big part of the sound making. Indeed, when we tried to loop our sound for the first time we realised that there was a big cut in the sound that was painfully annoying. to fix that issue we decided to mix the sound in audacity. This software was very useful when it came to smoothening the transition between sounds but also for keeping each sound effect at the same level of decibel.

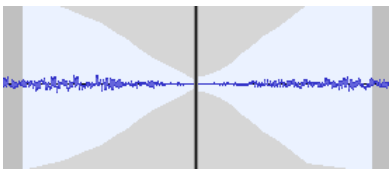


Figure 14: Sound edition

As you can see, (only one with sound) the clean transition of our gunshot, as we knew that this particular sound would loop a lot, so we made the transition between each gunshot clean.

5 Tim's progression

5.1 UI

To start with, we discussed some basic themes that would complement the rest of the game and decided to go with a futuristic, transparent blue concept so that the game scene would be visible through the canvas. After the first panel with buttons was created, it was very easy to create the others as we simply duplicated it and changed the text. Initially, we tried using a switch statement which opens the corresponding scenes using a tag. However, this method uses the fixed update function which is actually unnecessary and costly in terms of complexity. The solution was to use on click events which call the corresponding functions in the ButtonHandler.cs script which changes the scene using the index of scenes in the build settings. In terms of navigation, when the game first starts, a main menu scene opens which just contains a canvas. When either one of the buttons is pressed (Multiplayer or Single player), the right scene opens and the game starts. When the P key is pressed another menu comes up with two buttons; one for the controls manual and the other to go back to the previous menu.

5.2 HUD

Crosshair aim, health bar and offscreen enemy/missile indicator were the HUD features we wanted to implement. The easiest out of the three was the crosshair which simply consists of two ui images arranged in a triangle shape. Since the camera is fixed for the internal pilots view, the cross hair doesn't require any movement; however for the external view, the angle between the direction of the camera and the bullets projectile are too significant for a crosshair to be useful. Therefore, at the moment, it is only implemented on the internal view. Next, the health bar was relatively quick to set up. It consists of two ui images: the overlap and the RectTranform property of the bar set to middle stretch so that it can shorten itself only on the right side. A health variable used in another script allows us to vary the length of the health bar. Finally, the off screen indicator was the most difficult out of the three. In the beginning we tried to install a free unity asset and apply it to our scene. We found the manual quite vague and could not get it working on our own canvas. So we decided to write our own. This started with using ray casts from the centre of the canvas to the x,y position of the target on the canva plane. Since the canvas had to be in Screen Space - Overlay mode, its position doesn't change relative to the plane. Then we used the plane a reference, created a edge collider around the board of the canvas, found the vector direction to the target projected on the plane normal to the forward direction of the aircraft (local z axis) and then cast a ray from the centre of the canvas to in the correct direction and then use a RaycastHit to find the collision point to set the indicators position to. This became quite confusing and difficult to debug so we thought of an alternative and much simpler method. The local direction vector normalised multiplied by a constant which is added to the centre point of the canvas would give the position for the indicator around a circumference. After that, we changed the image to an arrow and added a line in the script that rotates it so that it points in the right direction using the inverse tan of the y component divided by the x component. Finally, when the target is in the camera's frame, we want the arrow to be disabled and instead a red translucent square to hover over the target. This took some time however was made very easy thanks to the WorldToScreenPoint

function provided by unity.

6 Terence's progression

6.1 Turret

On both the cargo plane and the little plane, a model of turret exists, however, it was not working properly for the first defence. From now on, every time a bullet will be fired, it will come out of the turrets. You can see in the following figure, the turret model we have designed.

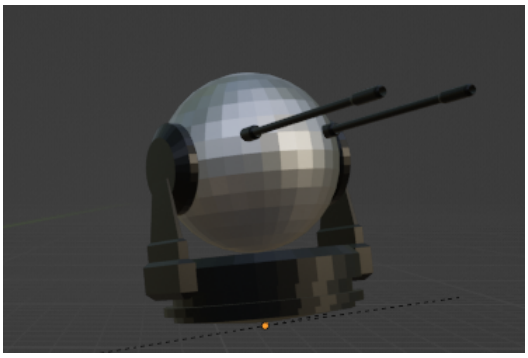


Figure 15: Turret model

This model was built to be animated, in fact the parenting allows unity to access and move different parts of this turret. Here, we make the sphere rotate on the Z axis so that the player can aim. The shooting system previously discussed in its own section allowed the turret implementation to be quick and easy. The only thing that had to be done was to create an empty object pointing to where ever the player wants to shoot. And the trick is done, the turret are now a feature of our game.

6.2 Maps

During this second period, we worked a lot on making the game look good. Because if the mechanics and the physics part count, the looking for a game is the first thing a player is going to enjoy. The map we showed during our first defence was not enough, it was just a test. For this second defence, we chose to implement two brand new and complete maps. Again, looking for realism we use google earth to find real places of the world and used these references to create our maps.

6.2.1 Forest map

The first map is: The Forest. For this one, we took our inspiration from a real forest in the north of Russia. You can see our inspiration comes from the Google Earth page in the following figure.

You can see in the middle of the map a take-off track. This is our central point. We have added a few planes so that it looks real. Everywhere else, the only thing you can see is forest, a lot of forest. The style of forest we have voted for is of the dense pine tree kind. This way, the player can enjoy good graphics while trying to destroy its enemy. To be more specific about the realisation of this map, we used a lot of paper and drawings to design this map. In fact, all of the tiles this map contains (81) were drawn beforehand on a piece of paper. It was useful to see clearly how every feature, the roads, the forest and the airfield would come together. Once this step was done, we started developing it inside Unity, and it was really quick to do since everything had already been thought of.

6.2.2 Desert map

We wanted the player to have a sense of immensity, to think that their plane could fly peacefully for hours in this somewhat empty desert, just before reminding them when they would fly over our crashed cargo plane that they were here to fight! Our desert maps may seem empty at first sight but it is far from being the case. Among other things, our second map features a crash site, a city, a broken airport, an oasis and even more to discover. The difficulty with this map was figuring out its scale because, like the other one, it is really big: 81 kilometers square. To make sure that the player understands how big this map is, we chose to put small details on each and every tile so that we never lose the sense of immensity that this map brings.

7 Conclusion

For the last defence, we had two plane models that both worked with the physics script. The first was a spitfire like design and the second a much larger cargo plane. The multiplayer was working allowing two players to dogfight on different computers. The first checkpoint for the AI was met which allows a plane to follow the player with some level of accuracy. A tutorial scene was created which consists of a very basic map and randomly generated balloons that explode on impact of the bullets or the plane itself. Finally, a basic woodland map was presented which stood as a prototype.

There has been plenty of progression since then. A third and final plane model has been created. The jet fighter was modelled using real plans and UV mapping for the textures. Sound effects have been added for the bullets, missiles, crashing of the plane and engine. The volume level of the engine sound increases as the throttle increases. Furthermore we have appropriate music playing in the background. The weapons system now consists of two different kinds of ammunition; bulletsttles and missiles. Bullets are instantiated and a ray cast in the forward direction of the plane checks if there was a hit on the target. The missiles can only be fired if it has locked on target and will follow the target plane unless "evasive" manoeuvres are performed. In terms of the AI, we have a script that follows the target with precision, rolls and pitches up if the pitch down angle is too great, can enter attack mode, chooses whether to shoot missiles or bullets and avoids the ground if necessary. To add more realism to the game, we have used particle effects to create trails at each wing tip of the flying planes, realistic smoke trails and an after burner for the jet fighter that increases in length when the throttle button is pressed. At this point, we now have two stunning maps with good quality clouds and textures. The previous woodland map has been improved to reduce lagging as well as randomly spawn trees. Our new desert map contains a cargo plane crash site and burning oil tanks that use the smoke trail particle effects, clouds, an oasis and a small town. Both of these terrains spread over a 81km square space allowing plenty of room to enjoy that gameplay. A very basic skeleton for the UI menu screens has been created with options to choose a game mode, mute the gameplay volume and view the control keys. We have also implemented a HUD that contains a crosshair for aiming the bullets, health bar and enemy indicator. The enemy indicator moves around the screen pointing at the position of the target while it is off screen and moves a red box over the target while the target is on screen. In order to easily share a demo of the game, we have added a compressed version to our website using simmer.io.

Certain aspects of the AI became increasingly difficult. This included ground avoidance since the script may want to perform manoeuvres that cause the plane to hit the terrain. This was solved with the use of raycasts directed 10 degrees below the forward direction of the plane and pulling up if the ground became too close. Aside from that, getting the AI controlled plane to fly smoothly without falling into a state where it overshoots the target.

The HUD enemy indicator was a rather straightforward idea however difficult to implement. The initial idea was to use ray casts shot from the centre with a direction vector that is the local displacement of the target/enemy plane projected onto the plane normal to the local z direction. This became quite difficult to debug so we found another much simpler way to do it using the same displacement vector normalised and multiplied by a radius constant which when added to the centre of the canvas position, gave the coordinates of the indicator.

The biggest problem that I encountered was with the particles effect, when trying to do the effects on the tip of the wings I ran into a problem that was the fact that the particles at the beginning of the effect were flickering so why was it doing it well it was because to simulate the trail I had ribbons connected between every particle so it wasn't working at the beginning because there were not always particle at the beginning meaning the ribbon only appears when a particles was created so to fix that I increase the number of particles per sec and increased the number of ribbons between particles to make it more smooth and enjoyable to watch.

For the modelling we struggled a lot with the uv mapping technique to apply the textures. The different faces can sometimes overlap one another with the automatic mode of blender and the jet fighter model

required us to do almost all of the uv mapping face by face. This is due to the fact that this plane has the greatest number of textures The only solution to that problem was repetitive hard work because we wanted the texture to be perfect. For the sound effect, the biggest problem was the way the sound broke each time it looped. This was really bad because it was really unpleasant to hear. We overcame this issue by using Audacity to edit the tracks and make them smooth at both the beginning and the end.

For the last defence, we aim to have our game completely finished. We will need to improve the UI visually as well as adding some extra features such as volume levels and being able to choose any map as well as any plane in the game. The tutorial scene that was created for the last defence will be added to the UI and will include some instructions. At the moment there are blank screens while scenes are loading so we decided that loading scenes will improve the players experience. We will build on the multiplayer adding another mode which allows a co-op scene against the AI. The AI will include some extra features such as energy management and different levels of difficulty. Since the missiles only fire when they are locked on, we will add another feature to the HUD so that the player knows when to shoot them. We will keep the website up to date and make some minor improvements. And finally, animations as well as extra drag will be implemented for the moving flaps of the planes.

Bibliography

[1] *(1087) How to make a Offscreen Target Indicator Arrow in Unity - YouTube*. https://www.youtube.com/watch?v=gAQpR1GNOOs&ab_channel=digijin. (Accessed on 04/24/2022).

[2] *Ace Combat 7: Skies unknown*. URL: https://en.wikipedia.org/wiki/Ace_Combat_7:_Skies_Unknown.

[3] *Air combat*. URL: https://en.wikipedia.org/wiki/Air_Combat.

[4] *Aircraft principal axes*. URL: https://en.wikipedia.org/wiki/Aircraft_principal_axes.

[5] *Bartendu*. URL: https://areas0.github.io/website/index.html.

[6] *Blender 3D modeling : The ultimate collection*. URL: https://conceptartempire.com/blender-modeling-tutorials/.

[7] *Create a map in unity*. URL: https://docs.mapbox.com/help/tutorials/create-a-map-in-unity/.

[8] GameDevLessons. *Flight SIM control, Terrain Basics, Chase Cam, skybox*. URL: https://www.youtube.com/watch?v=lCulq9J0Y9E.

[9] *GitLab*. URL: https://en.wikipedia.org/wiki/GitLab.

[10] *Gitlab Documentation*. URL: https://docs.gitlab.com/.

[11] *Host (network)*. URL: https://en.wikipedia.org/wiki/Host_(network).

[12] *Local area network*. URL: https://en.wikipedia.org/wiki/Local_area_network.

[13] Renan Oliveira. *How to create a multiplayer game in Unity*. URL: https://gamedevacademy.org/how-to-create-a-multiplayer-game-in-unity/.

[14] Dwight Pavlovic. *Video game genres : HP® Tech takes*. July 2020. URL: https://www.hp.com/us-en/shop/tech-takes/video-game-genres.

[15] Unity Technologies. *Setting up Unity Multiplayer*. URL: https://docs.unity3d.com/Manual/UnityMultiplayerSettingUp.html.

[16] Unity Technologies. *Unity Collaborate*. URL: https://unity.com/fr/unity/features/collaborate.

[17] *Unity - Manual: Unity User Manual 2021.3 (LTS)*. https://docs.unity3d.com/Manual/index.html. (Accessed on 04/24/2022).

[18] *Unity - Scripting API: Renderer.isVisible*. https://docs.unity3d.com/ScriptReference/Renderer-isVisible.html. (Accessed on 04/24/2022).

[19] *Unity - Scripting API: SceneManagement.SceneManager.LoadScene*. https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html. (Accessed on 04/24/2022).

[20] Vazgriz. *Creating a Flight Simulator in Unity3D*. URL: https://vazgriz.com/503/creating-a-flight-simulator-in-unity3d-part-3/.

[21] *Yughues Free Sand Materials | 2D Floors | Unity Asset Store*. https://assetstore.unity.com/packages/2d/textures-materials/floors/yughues-free-sand-materials-12964. (Accessed on 04/24/2022).