# IMPERIUS

A 3D SPACE REAL TIME STRATEGY GAME

Submitted to the Faculty of

**WORCESTER POLYTECHNIC INSTITUTE**

in partial fulfillment of the requirements for the degree of

*BS: Computer Science*

*BA/BS: Interactive Media and Game Development*

Authors:

**Matthew Hendrickson** mchendrickson@wpi.edu

**Kurtis Kiai** kmkiai@wpi.edu

**Aaron Waldman** awaldman@wpi.edu

**Benjamin Martin** btmartin@wpi.edu

Faculty Advisors:

**Prof. Ben Schneider** bschneider@wpi.edu

**Prof. Joseph Beck** josephbeck@wpi.edu

## 1. Abstract

*Imperius* is a 3D real-time strategy space game developed in the Unity Engine by a team of four WPI seniors over the course of the 2022-2023 academic year. The game features a 6-mission campaign and 4-player Steam integrated multiplayer. The development team employed common methodologies such as scrum and agile to stay on track and meet deadlines, while also adapting to the challenges of designing a complex game. In addition to the core team, two other WPI students contributed through independent study projects, and around two dozen voice actors were also utilized. This project represents the culmination of several years of work in a variety of different disciplines such as Computer Science, Software Engineering, 3D modeling, SFX, VFX, Level Design, and more.

## 2. Acknowledgements

The project team acknowledges the contributions of several exceptionally talented individuals who played vital roles in the development and success of the game. First and foremost, the team expresses sincere gratitude to Professor Benjamin Schneider for his invaluable expertise in narrative design, RTS design, and team management, as well as Professor Joseph Beck for his abundant programming knowledge and technical know-how. Their feedback and support throughout the year-long development process were essential to the team's success. Additionally, Mason Moore's exceptional skills in creating the game's cutscenes and trailers, Caleb Scopetski's impressive work in researching and developing the game's AI, and Eric Zhu's consistent support, feedback, and overall input from the project's outset were crucial to the project's success. The team would also like to extend appreciation to the over two dozen voice actors whose contributions brought the narrative to life, and Varun Bhat for his support in recording them.

*Imperius* would not have been the same without the dedication and contributions of these exceptional individuals.

## Contents

# 4. Introduction

Imperius is a 3D space-themed real-time strategy (RTS) game developed by a team of WPI students, originally started as a hobby project in the Fall of 2019. The game features mechanics inspired by popular RTS games like Total War, Star Wars: Empire at War, and Halo Wars. The full game, including 6 unique missions, is expected to take around 1-2 hours to finish and includes 2 cutscenes. Audio from almost 2 dozen voice actors was utilized to bring a 60-page narrative to life. It also has a multiplayer skirmish mode that supports up to 4 players, with Steam integration.

The development of Imperius involved a wide range of skills, including programming, audio design, 3D modeling, VFX, level design, texturing, UI design, and more. Advanced programming techniques such as Octrees for pathfinding, multithreading, network engineering, and more were utilized to provide an efficient and smooth gameplay experience. The porting of the game from 2.5D to 3D and the organization of the massive codebase alongside implementing outside APIs were also significant challenges that the team had to overcome.

The background section of the paper discusses the challenges of the project and provides information about the team's research and playtesting. The design section of the paper covers the concepts and rationale for the mechanics and other critical elements of the game. The audio and art development sections detail the challenges faced and decisions made throughout development, including the use of Unity's particle system for VFX and the creation of detailed textures for the game's units and environments. The narrative section dissects the 6 campaign missions and 2 cutscenes detailing the brutal war between the militarist Imperius and the zealous Divinity. The team also conducted playtesting sessions to gather feedback on gameplay, balance, and overall user experience. The results and data section of the paper documents the findings from these sessions and how they influenced the team's decision-making process.

Overall, Imperius represents the culmination of the team's skills and efforts and showcases their ability to create a unique, high-quality, 3D RTS game that can compete alongside other games in the genre.

# 5. Background

## 5.1 Pre-MQP History

In late 2019, the original creator involved in the development of Imperius conceived the idea of recreating *Star Wars: Empire at War* utilizing community assets. Originally named *Imperium*, (Figure 1) the project was halted after a year of development due to a lack of experience and time.

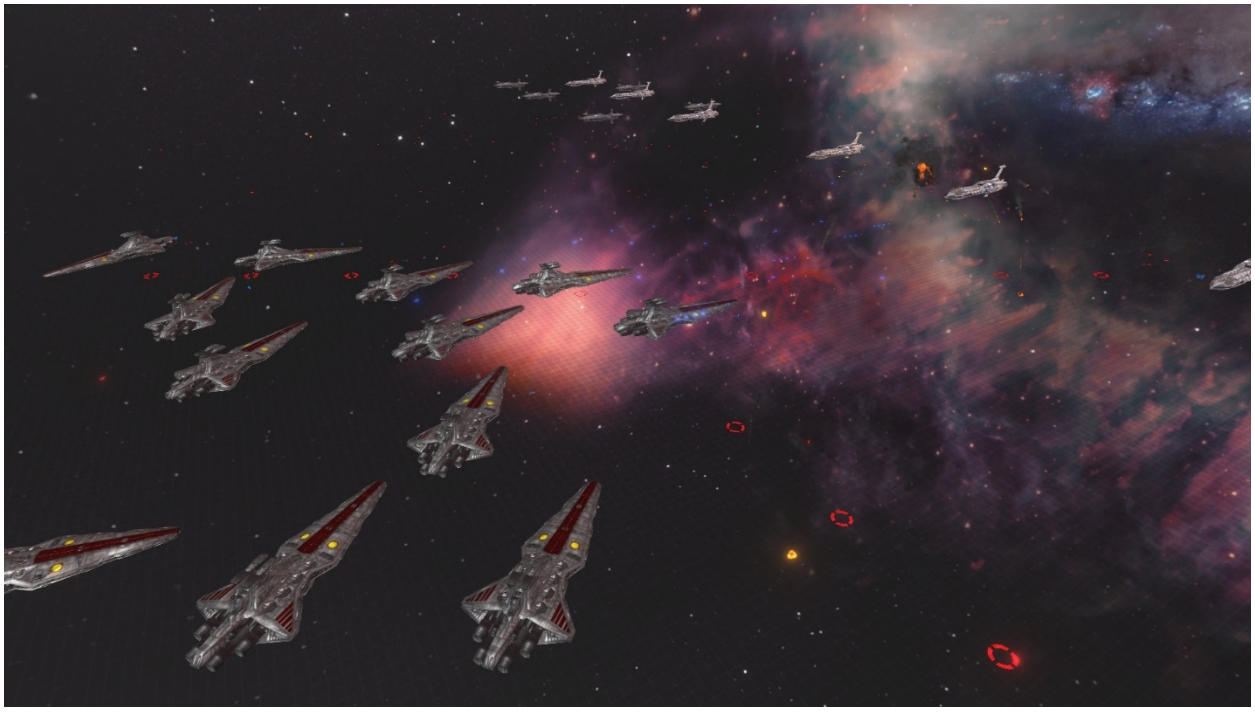


**Figure 1: Imperium Pre-Alpha Build 0.4**

However, the project was revived under the name *Imperius* and was developed for another six months for course credit at a different university. When the senior transferred to WPI, the project was finally completed as a yearlong MQP.

## 5.2 Game Inspirations

A Real-Time Strategy (RTS) game is a type of strategy video game where players oversee building and managing resources, bases, and armies in real-time, typically viewed from a top-down perspective. Players must make strategic decisions quickly, as the game progresses in real-time, and adapt to changing circumstances as they arise. RTS games typically involve a mix of resource management, base building, and combat, and often require players to gather resources, construct buildings and units, and engage in battles with opposing players or computer-controlled enemies. Examples of popular RTS games include Command and Conquer, StarCraft, and Halo Wars.

### 5.21 Star Wars: Empire at War

*Star Wars: Empire at War* (Figure 2) is a real-time strategy game set in the Star Wars universe developed by Petroglyph Games in 2006. One of its key gameplay mechanics is the use of hardpoints, which are specific locations on ships or structures that can be targeted and destroyed. Hardpoints can vary in function, from weapons that can be taken out to reduce enemy firepower, to subsystems like engines or shields that can cripple a ship's ability to maneuver or defend itself.

**Figure 2: Star Wars: Empire at War Cover Art**

Players must strategize carefully when attacking or defending hardpoints, as taking out critical ones can give a significant advantage in battle. This system adds an extra layer of depth to *Empire at War's* already complex gameplay, making battles feel more dynamic and engaging. We designed a heavily modified version of this hardpoint system to make our gameplay significantly more engaging.

### 5.22 Command and Conquer

*Command and Conquer* (Figure 3) is a real-time strategy game series that has been popular since its initial release in 1995. The series takes place in a fictional world where the player must build and maintain a base, gather resources, and amass an army to defeat enemy factions. The game's mechanics revolve around resource management, unit control, and strategic decision-making.

**Figure 3: Command and Conquer Remastered Cover Art**

Players must balance their economy, production, and defense while strategically deploying their units to outmaneuver and overpower their opponents. Each faction in the game has unique units and technologies, allowing players to develop different strategies and approaches to each battle. Overall, *Command and Conquer*'s addictive gameplay mechanics, immersive storyline, and innovative unit designs have made it a beloved classic among strategy game enthusiasts and was a major inspiration for many of the mechanics used in *Imperius*.

## 5.23 Halo Wars

*Halo Wars* (Figure 4) is a real-time strategy game set in the Halo universe, released in 2009. The game differs from other traditional RTS games in that it was designed specifically for consoles and therefore features simplified controls and gameplay mechanics. The player assumes the role of a commander in charge of the UNSC forces, managing resources, constructing buildings, and recruiting units to engage in battles against the Covenant. Gameplay mechanics are centered around base building, resource management, and unit management, as players must balance their economy, research new technologies, and develop their armies to outmatch their opponents.

Figure 4: Halo Wars Cover Art

Each faction in the game has unique units and abilities, allowing players to experiment with different strategies and approaches to each battle. Halo Wars also features a robust single-player campaign with a rich story that expands upon the lore of the Halo universe. The game's simplified controls and mechanics make it accessible to new players while still providing a challenging and engaging gameplay experience for RTS veterans. We wanted to keep the same sense of simplicity while offering a wide variety of mechanics to allow for greater strategic decisions.

## 5.24 Homeworld

*Homeworld* (Figure 5) is a real-time strategy game that was released in 1999, known for its 3D graphics and unique gameplay mechanics. The game is set in space, and players control a fleet of ships as they navigate through the galaxy, attempting to reclaim their home planet. The game features a non-linear storyline, with each mission having multiple objectives that the player can complete in any order. One of the unique mechanics of the game is the ability to move ships in all three dimensions, allowing for complex tactical maneuvers and flanking strategies.

**Figure 5: Homeworld Cover Art**

Homeworld also features a deep unit customization system, with the ability to research new technologies and build different types of ships with unique abilities. Overall, Homeworld is praised for its immersive storyline, innovative gameplay mechanics, and stunning visuals, making it a classic in the real-time strategy genre. We primarily took inspiration from the 3D elements Homeworld offers and improved upon them to create more innovative gameplay.

## 5.3 Narrative Inspirations

### 5.31 Real World

#### *5.311 Dehumanization and Superiority*

The first major point of contention to be found in all these stories of fascism starts with the dehumanization of their victims. Hitler spoke of the Jews in one speech as the greatest threat to western culture and in another as the weakest race on the planet. (Hitler, 1939) This cognitive dissonance was fully exploited amongst the soldiers to create both a sense of moral superiority and urgency that only violent action could rectify.

## *5.312 Abuse, Manipulation, and Radicalization*

In Imperial Japan, soldiers were molded from childhood to be obedient and hyper-focused on the conquering of Asia. They expected to be treated as rightful rulers in China since they believed Japanese citizens were of a higher social caste than Chinese citizens. The commonly accepted explanation of the atrocities committed by Imperial Japan is that their soldiers held such a low opinion of themselves that, when given positions of authority, they projected all their pent-up self-loathing on others and lashed out at them. (Chang, 1997)

*"Duty is heavier than a mountain; death is lighter than a feather"* was Imperial Japanese Army motto.

The Nazi party had similar motivations. From birth, infants have an innate tendency to form a bond with their primary caregivers. The Nazis aimed to create children who were resilient, unfeeling, lacking in empathy and had minimal emotional connections with others. They recognized that by withholding affection, they could achieve this objective. (Kratzer, 2019)

## *5.313 Denial and a lack of accountability*

The command structure of the Nazi hierarchy, specifically the SS, provided soldiers with mostly guiding principles. Junior officers were given the freedom to take independent action to achieve their intended objectives, which proved disastrous to the civilian populations under their boots. The Nuremberg trials led to much finger-pointing as officers often used the Nuremberg Defense of: "I was just following orders" to justify their actions. Much blame was put on the most powerful members of the Nazi party, who had conveniently died or committed suicide. In this way, the former Nazi officials did everything they could to deny anything that had weak evidence and shift blame to others anything substantially backed with evidence. (Priemel, 2018) As lies were exposed and evidence came to light, it was clear that these soldiers had a moral obligation to disobey their superiors yet repeatedly decided not to.

## *5.314 Citations*

Kratzer, Anne. "Harsh Nazi Parenting Guidelines May Still Affect German Children of Today." *Scientific American*, Scientific American, 4 Jan. 2019, https://www.scientificamerican.com/article/harsh-nazi-parenting-guidelines-may-still-affect-german-children-of-today1/.

Chang, Iris. *The Rape of Nanking: The Forgotten Holocaust of World War II*. Basic Books, 1997.

Priemel, Kim Christian. *The Betrayal: The Nuremberg Trials and German Divergence*. Oxford University Press, 2018.

Hitler, Adolf. "Address to the Reichstag." 30 January 1939, Reichstag, Berlin.

"Why Did the Holocaust Happen?" *Radicalization of the Administration of the Nazi State – The Holocaust Explained: Designed for Schools*, https://www.theholocaustexplained.org/how-and-why/why/radicalisation-of-nazi-administration/.

### 5.32 Fantasy

#### *5.321 Star Wars*

Star Wars is a beloved franchise that has had a significant impact on both the film and gaming industries. The world-building of Star Wars includes a rich and diverse universe with a variety of planets, species, and technologies. The iconic space combat featured in Star Wars has been particularly influential in the gaming industry, with many games drawing inspiration from the franchise's starfighter battles. The use of distinct factions, such as the Rebel Alliance and the Galactic Empire, also had an impact on the development of real-time strategy games with many utilizing similar faction-based mechanics. Additionally, Star Wars has influenced the use of sound design in games, with the iconic sound effects and music of the franchise becoming synonymous with the sci-fi genre. Overall, the world-building and space combat of Star Wars has left a lasting impact on the gaming industry, inspiring countless games and contributing to the growth and evolution of the genre. We took inspiration from Star Wars by drawing on its space combat mechanics and world-building to help improve our ideas.

## 5.4 Art Inspirations

### 5.41 Imperius

To decide how to prototype the new Imperius ships, we researched other popular games and movies which we believed had very prominent and unique designs. As we ventured through popular space-related titles, we were left with three which would be used to influence the art style.

#### *5.411 Eve Online*

Eve Online is an MMO-RPG (Massively Multiplayer Online Role-Playing Game) that takes place in the world of New Eden, a cluster of stars and planets which are inhabited by humans. In New Eden, there are multiple factions with a variety of ships, abilities, and individual art styles.

It’s common for ships in Eve Online to follow the simple rule that larger ships prevail over smaller ships. The notoriety of larger ships turning the tides of battle gave a sense of awe. A fundamental objective in the design of the larger vessels within the Imperius fleet was to adhere to this principle, to create an exhilarating experience for players upon witnessing massive dreadnoughts enter the fight.

#### *5.412 Warhammer 40K*

Warhammer 40K is a miniature wargame based on multiple science-fiction novels. The series is set in the distant future, with multiple races fighting in an inter-galactic war. When prototyping the art style for the Imperius, one of the art directions of interest was gothic sci-fi as shown in .  Warhammer 40K heavily explored this art style and most of its ships, as seen in Figure 7, follow the gothic sci-fi art direction.

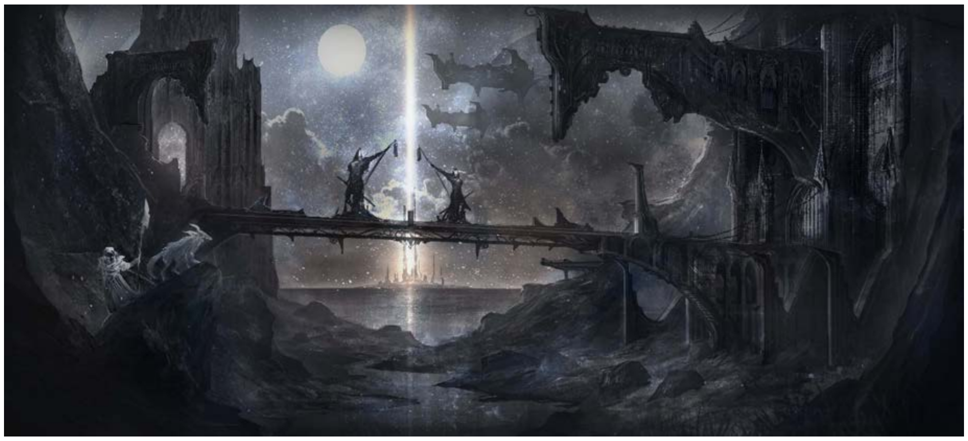


**Figure 6: An Illustration of the Gothic Sci-fi Art Style**

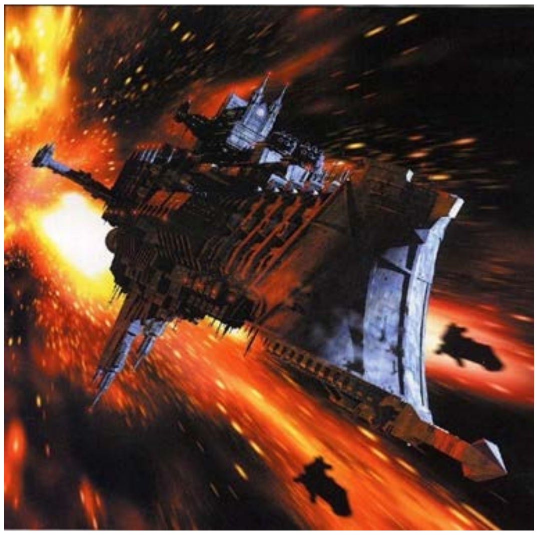


**Figure 7: A Battleship from the Warhammer 40K Universe**

### *5.413 Star Wars*

One of the Star Wars ships that served as an inspiration for our game design was the Xyston Class Star Destroyer shown in Figure 8. This vessel exudes a sense of power, with its streamlined body and towering bridge situated at the stern, which seems to dominate the entire ship. The sharp, pointed appearance and the formidable, single cannon at the bow contribute to the ship's menacing aura. We aimed to imbue our largest Imperius ships with a similar sense of superiority and authority, such that they would project a sense of dominance over everything else on the battlefield.

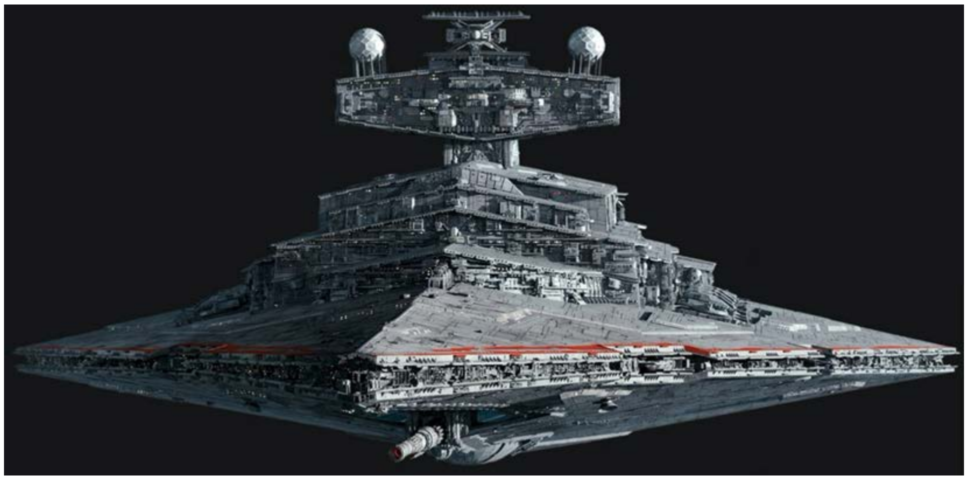


Figure 8: A Xyston-class Star Destroyer from the Star Wars Franchise

## 5.42 Divinity

The Divinity faction in the game serves as the main antagonist and was designed with a different art direction than the Imperius. While the Imperius was inspired by Star Wars and had a powerful, sleek look, the Divinity was meant to have a more royal appearance, with ships that convey a sense of alien architecture. The inspiration for the Divinity came from the Covenant enemy force in the Halo universe, which is known for its ornate and intricate designs. Additionally, the Divinity was also influenced by an unknown alien civilization, which gave the art direction a unique and mysterious quality.

### *5.421 Halo*

Halo is a fictional universe that chronicles the adventures of Master Chief, a highly skilled super soldier, and his trusted A.I. companion, Cortana. Together, they embark on a perilous journey across the cosmos to combat The Covenant: a group of extraterrestrial invaders who are zealously attempting to find salvation. This salvation dubbed “The Great Journey” is to locate and activate the Halo rings, which have the catastrophic power to obliterate all life within the galaxy.

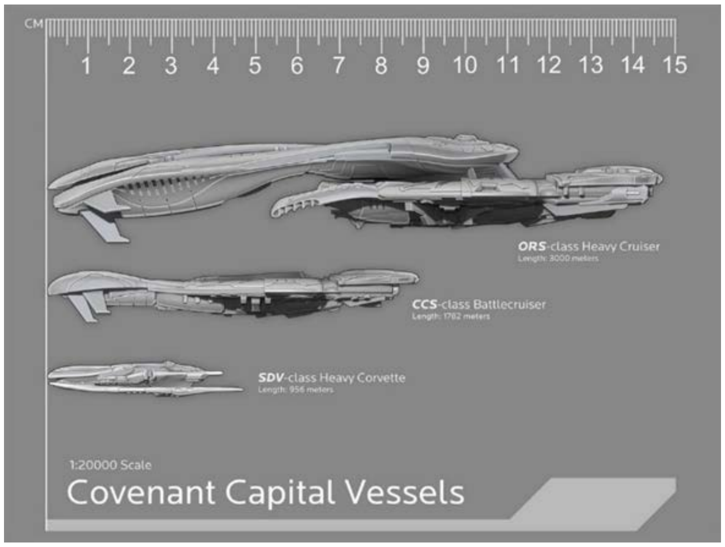


**Figure 9: Covenant Ship Designs from the Halo Series**

As shown in Figure 9, the art style used for the Covenant presented a distinctive perspective on how an alien spacecraft should appear, featuring non-uniform, asymmetrical shapes, and an unbalanced visual weight. The Covenant vessels boast a sleek, mystical design, giving a sense of otherworldliness. In designing the Divinity's technology, we aimed to create a similar sense of intrigue and enigma, with their ships featuring unrealistic visual balancing and weight proportions. Moreover, we envisioned the Divinity as utilizing archaic sources of energy to power their structures and vessels, further contributing to their mysterious nature.

## 5.5 User Interface Design

The user interface (UI) is how players interact with a game during play. For Imperius, the goal was to capture the science fiction ambiance of the game while maintaining visual clarity through a simple and minimalist design. The type and amount of information displayed on screen varies significantly across different game genres, necessitating a tailored approach to UI design. Our exploration of other popular real-time strategy games was aimed at determining the type of UI that Imperius required. To gain insight into what constitutes an effective UI, we first examined known UI design errors in the field. Our initial research focused on the UI of Eve Online as seen in Figure 10, which is infamous for its “spreadsheet simulator”. Although the design of Imperius units drew inspiration from the models used in Eve Online, we aimed to avoid its UI pitfalls.

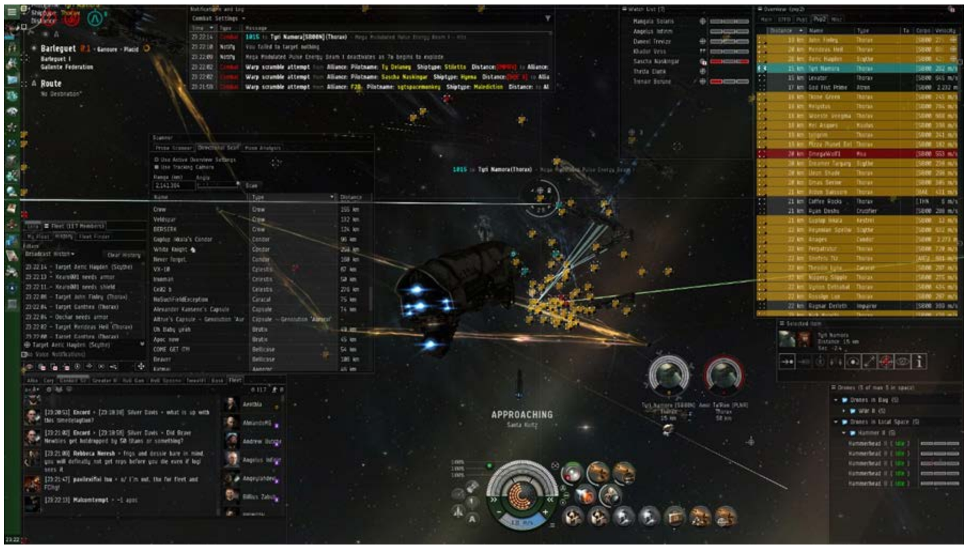


Figure 10: An Example of the User Interface in Eve Online

After researching various games, we decided to model Imperius' user interface after that of Halo Wars 2. This was based on the fact that Halo Wars 2 features a minimalist UI design as seen in Figure 11. The user interface showcases the game's stunning graphics, while also being a real-time strategy game set in the science fiction genre, which aligns with Imperius' thematic vision. We, therefore, took the UI of Halo Wars 2 as a starting point in developing Imperius' user interface.

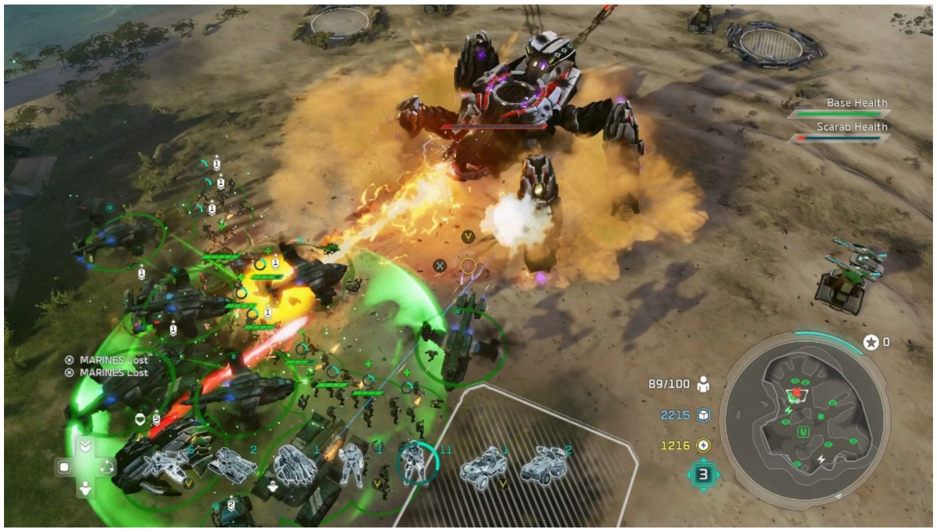


Figure 11: Halo Wars 2 User Interface

## 5.6 Music Inspirations

We drew inspiration from many different sources when composing music for Imperius. One of our primary influences was the orchestral battle music found in the works of *Two Steps From Hell* and the soundtrack of the game *Nier: Automata*. Many of the compositions from these sources employed a trumpet lead melody, quick violin arpeggios, and timpani and military snare hits for the drums. We incorporated these key elements into our compositions to create energetic, grand-sounding battle songs.

In addition to orchestral music, we also took elements from synth-wave artist *Carpenter Brut* and the general synth-wave trend of the 1980s. We integrated classic 80s synths, a steady kick drum rhythm, and sidechaining the synth chords to the kick drum to create a pulsating, breathing effect in our synth-focused tracks. This helped give our soundtrack a distinctive sci-fi aesthetic.

To bring more variety to the soundtrack, we created remixes of most of our original songs in the style of certain percussion-focused buildup tracks from the soundtracks of real-time strategy games *Command & Conquer 3* and *Halo Wars 2*. These remixes incorporated echoing hi-hats that pan left and right, aggressive bass guitar, and minimalist chords and melody to produce a more subtle but still dramatic musical experience. Overall, we are satisfied with the final soundtrack, as it provides a suspenseful and thrilling backdrop to the game.

# 6. Game Implementation

## 6.1 Code Refactoring

The previous prototypes of *Imperius* utilized legacy code and APIs from 2019, which posed a significant challenge in upgrading the project to the current 2021 Long-Time Service (LTS) version of Unity. Extensive efforts were required to update the existing codebase and ensure its compatibility with the newer version of Unity. Additionally, significant tech debt had arisen due to poor coding practices which required refactoring. Thus, a significant amount of time was devoted to ensuring that the codebase adhered to standard Object-Oriented Programming (OOP) design patterns, while also optimizing it to improve resource efficiency. The refactored UML diagram is shown in Figure 12.

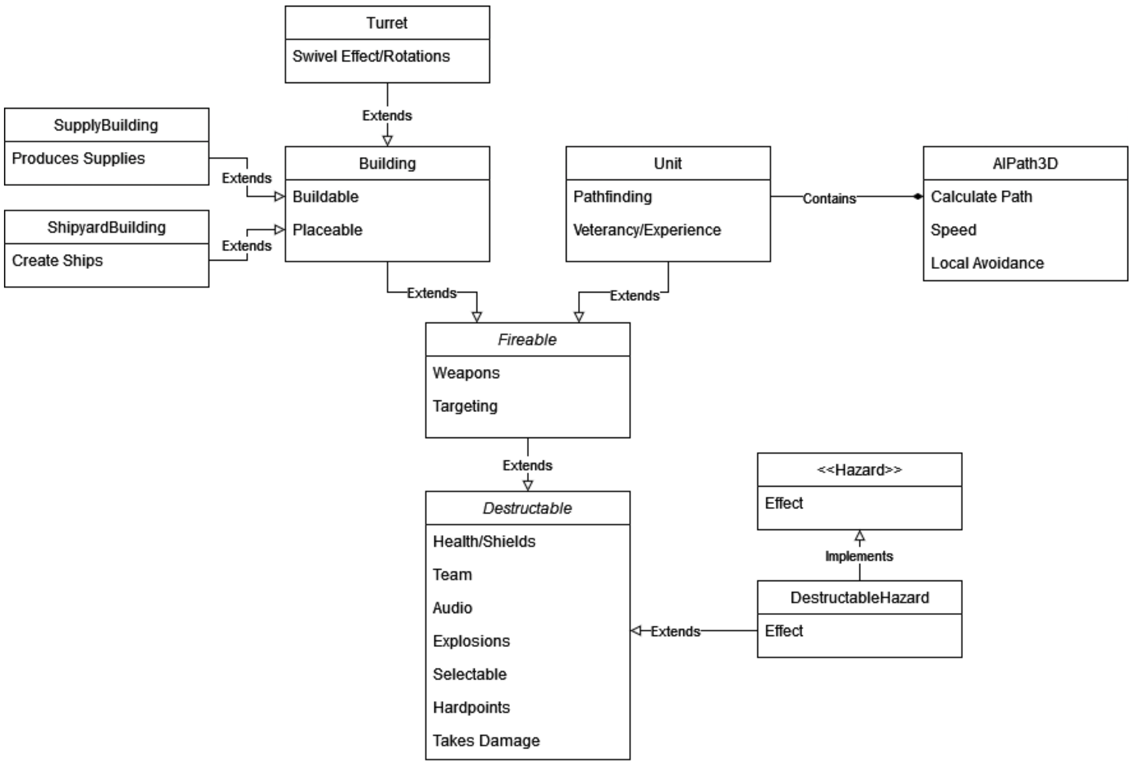


**Figure 12: UML Diagram of Reorganized Imperius Unit Methods**

## 6.2 Units

There are a total of 24 different types of Units in *Imperius*, each crafted to fill a particular role. For the Imperius faction, units were designed with a theme of aggression and cunning, with a heavy focus on damage and destruction above all else. They appropriate and use any tools they can to get an edge, such as stealing cloaking and FTL inhibitor technology from various alien races and using them for their own purposes. Figure 13 describes each Imperius Unit and its relevant information.

| *Name* | *Unit Type* | *Role* | *Description* |
|---|---|---|---|
| *Espion* | Patrol Craft | Scout | Standard fighter, cheapest craft, can suicide |
| *Nerva* | Corvette | Scout/Anti-Fighter | Heavy scout, excels in fighter combat |
| *Elysium* | Frigate | Debuff/CC | Long-range unit that strips shields |
| *Lodestar* | Destroyer | Support/CC | Has the ability to allow units to FTL to it |
| *Sentinal* | Destroyer | Anti-Fighter, DPS | Anti-fighter role, used to protect larger ships |
| *Noctus* | Stealth Destroyer | Stealth | Ambush using invisibility |
| *Garuda* | Carrier | Carrier | Holds fighters and has a complement of missiles |
| *Calibre* | Heavy Cruiser | DPS/CC | Long range, high damage, low health |
| *Verdancy* | Battle Cruiser | DPS/TANK | Front line unit, moderate damage, high health |
| *Aurelius* | Battleship | Support/DPS/CC | Slow, late-game frontline, high damage, health, AOE. |
| *Solace* | Battleship | Support/CC/Heal | Slow, heals and buffs units close to it, low armament |
| *Pacific* | Dreadnought | Super Unit | Super unit, slow speed, high health, high damage |

**Figure 13: List of Imperius Units**

Units in the Divinity were designed with a more eclectic role, having more radical and ethereal abilities. There is also a sense of desperation in the Divinity's tactics, as they are losing the war. Figure 14 describes each Divinity Unit and its relevant information.

| *Name* | *Unit Type* | *Role* | *Description* |
|---|---|---|---|
| *Argo* | Patrol Craft | Scout | Standard fighter craft, fast, weak, nimble |
| *Nile* | Corvette | Scout/Anti-Fighter | Heavy scout, anti-fighter capabilities |
| *Ishim* | Frigate | Anti-Fighter/Anti-Stealth | Anti-fighter, anti-stealth |
| *Sladen* | Destroyer | Support | Fires low-velocity volatile plasma spheres |
| *Censeur* | Destroyer | Support/CC | Support unit with CC abilities |
| *Zaraton* | Carrier | Tank | Used for soaking up damage from other ships |
| *Castilian* | Cruiser | Buff/Support | Generalist, all-around support ship |
| *Taniwa* | Cruiser | Buff/Support | Slowly repairs health and increases armor |
| *Aeneas* | Battlecruiser | Glass Cannon | Literally a giant laser on a fuel tanker |
| *Seraphim* | Battleship | DPS | Standard Divinity battleship |
| *Polyanthus* | Battleship | Support/DPS | Large support battleship |
| *Akuma* | Super Unit | Super Carrier | Mobile shipyard, super unit |

**Figure 14: List of Divinity Units**

Each ship has a specific role, and some have variants depending on various factors. (Figure 15)

| *Size* | *Name* | *Variants* | *Rough Size (m)* |
|---|---|---|---|
| *1* | Patrol Craft | | <50 |
| *2* | Corvette | | 50-100 |
| *3* | Frigate | | 100-200 |
| *4* | Destroyer | Stealth | 200-400 |
| *5* | Carrier | Light, Heavy | 400-800 |
| *6* | Cruiser | Light, Heavy, Battle | 400-1000 |
| *7* | Battleship | Flagship | 1000-1200 |
| *8* | Dreadnought | | >2000 |

**Figure 15: List of Unit Types**

## 6.21 Weapon Rotation Locks

In the context of naval warfare, it is a well-established principle that a ship's turrets must only fire in directions where the hull of the vessel does not intersect their line of fire. However, when it comes to real-time strategy (RTS) games, the application of this tactic presents a range of advantages and disadvantages, particularly in a 3D environment. On the one hand, allowing turrets to fire through a ship's hull can broaden their firing arc, providing a greater capacity for engaging multiple targets and potentially increasing the overall effectiveness of the ship. However, such a tactic can undermine the player's sense of immersion in the game, and limit the range of tactical options, including flanking maneuvers. Additionally, when firing arcs are too small, it can create logistical difficulties for positioning units effectively. To find a balanced solution, we opted for a compromise that restricts the XZ rotation (left/right) to specific angles but lets the XY rotation (up/down) have a free range of motion. This approach balances the positives and minimizes the negatives of each potential solution. An example of this is shown in Figure 16.

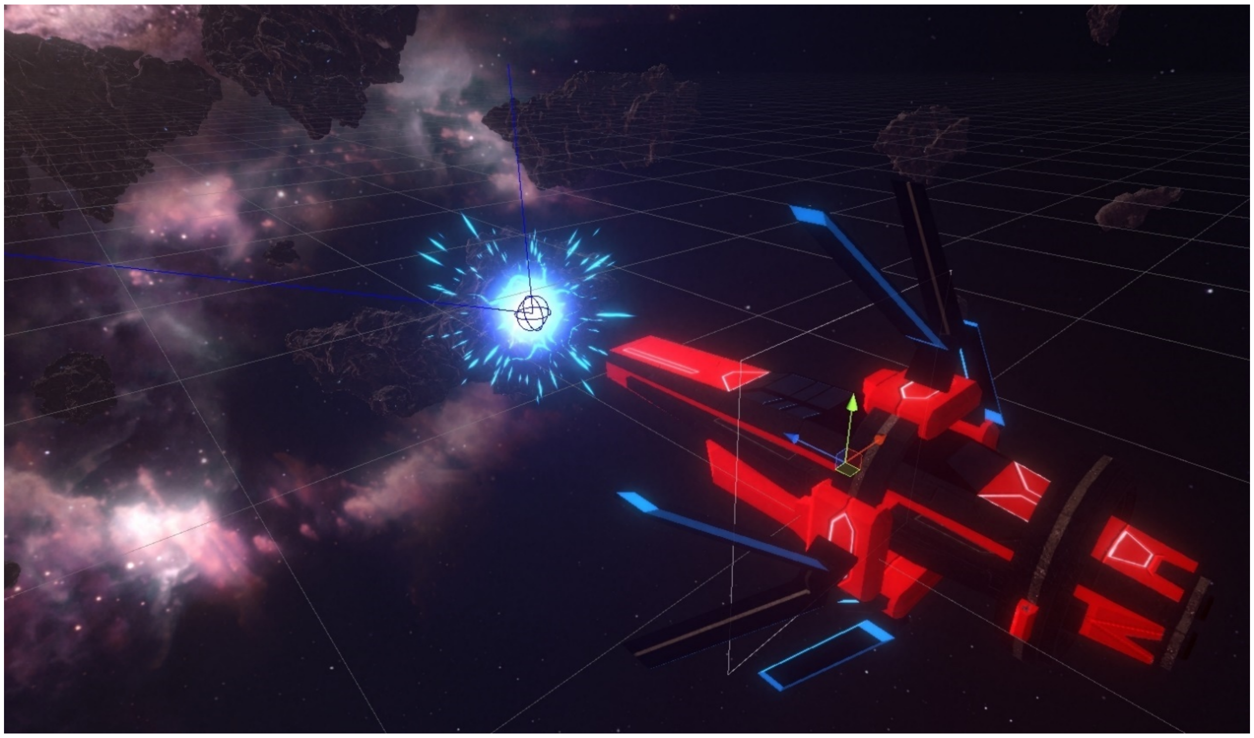


**Figure 16: The Firing Arc of an Imperius Ship**

The primary challenge associated with implementing this system centered around ensuring that any rotations applied to the parent object did not disrupt the rotations to the weapon. Several iterations and fine-tuning adjustments were necessary before the system functioned correctly. The resulting system effectively enhanced gameplay by making unit positioning a crucial strategic element, further emphasizing the importance of tactical planning and execution in the game. Figure 17 describes the code that enabled the mechanic in Figure 16.

```
function AngleBetween(target, angle1, angle2)
    while angle1 < 0:
        angle1 = angle1 + 360
    while angle2 < 0:
        angle2 = angle2 + 360
    while target < 0:
        target = target + 360

    rAngle = ((angle2 - angle1) % 360 + 360) % 360
    if rAngle >= 180:
        temp = angle2
        angle2 = angle1
        angle1 = temp

    if angle1 <= angle2:
        return target >= angle1 AND target <= angle2
    else
        return target >= angle1 OR target <= angle2
```

**Figure 17: Angle Between Pseudocode**

Where the target angle is calculated as:

$$atan2(enemyVector.y - weaponVector.y,\ enemyVector.x - weaponVector.x) * \frac{180}{\pi}$$

## 6.22 Hardpoints

The real-time strategy game Star Wars: Empire at War introduced an innovative system known as hardpoints. Hardpoints refer to subsystems of a ship such as weapons, engines, or shields that can be targeted and destroyed, resulting in significant performance degradation to the overall unit. We incorporated this system (Figure 18) using a simple algorithm that identifies the closest hardpoint to the point of collision, allowing the player to simultaneously damage both the hardpoint and the attached unit. Although Unity's particle system did not natively support this feature, we developed a listener design pattern that enabled each particle to reference and damage all colliders in the scene appropriately. This approach resulted in an immersive and challenging gameplay experience that added a new level of realism to our game.

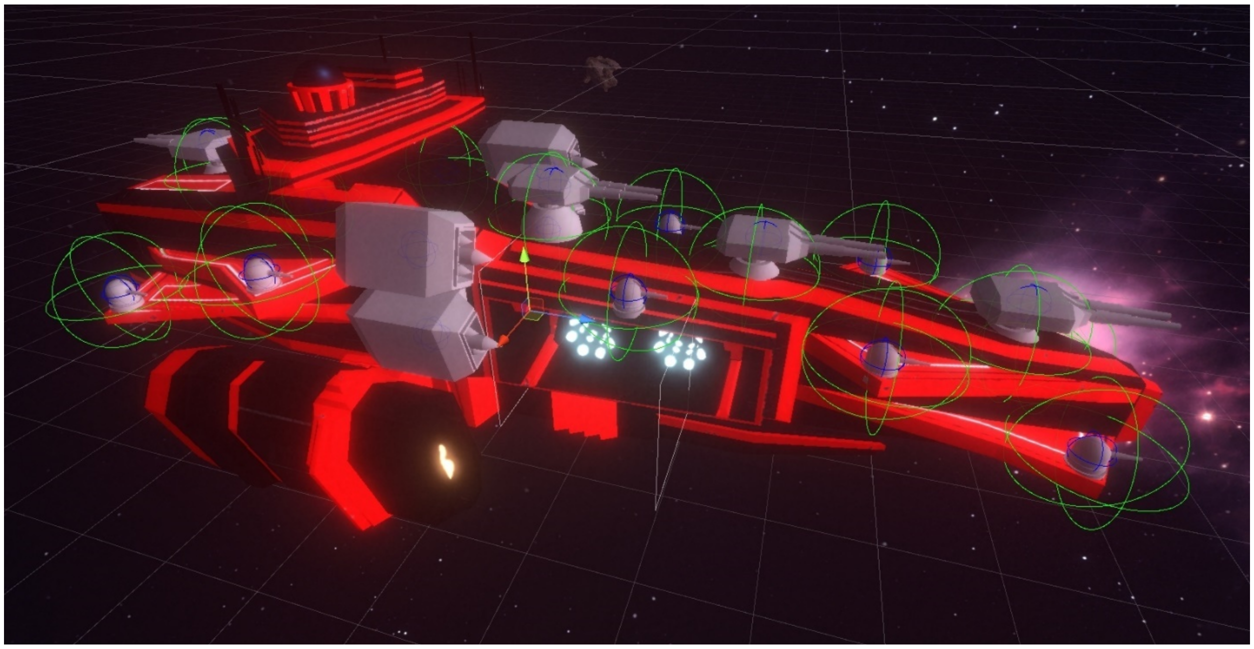


**Figure 18: An Example of Hardpoints**

## 6.23 Unit Lines

During the Revolutionary War, soldiers would often line up neatly in rows and fire at each other. This was a common strategy in military warfare, especially when engaging enemies who are in open spaces or on flat terrain. By lining up soldiers in a straight line and firing in unison, it allowed for maximum firepower to be concentrated on a specific area, increasing the likelihood of defeating the enemy. This tactic led to the design of a mechanic in RTS games to quickly and efficiently group and rotate units at desired locations. (Figure 19) It can also be used to protect weaker units that are positioned behind the frontline, such as archers or siege weapons.

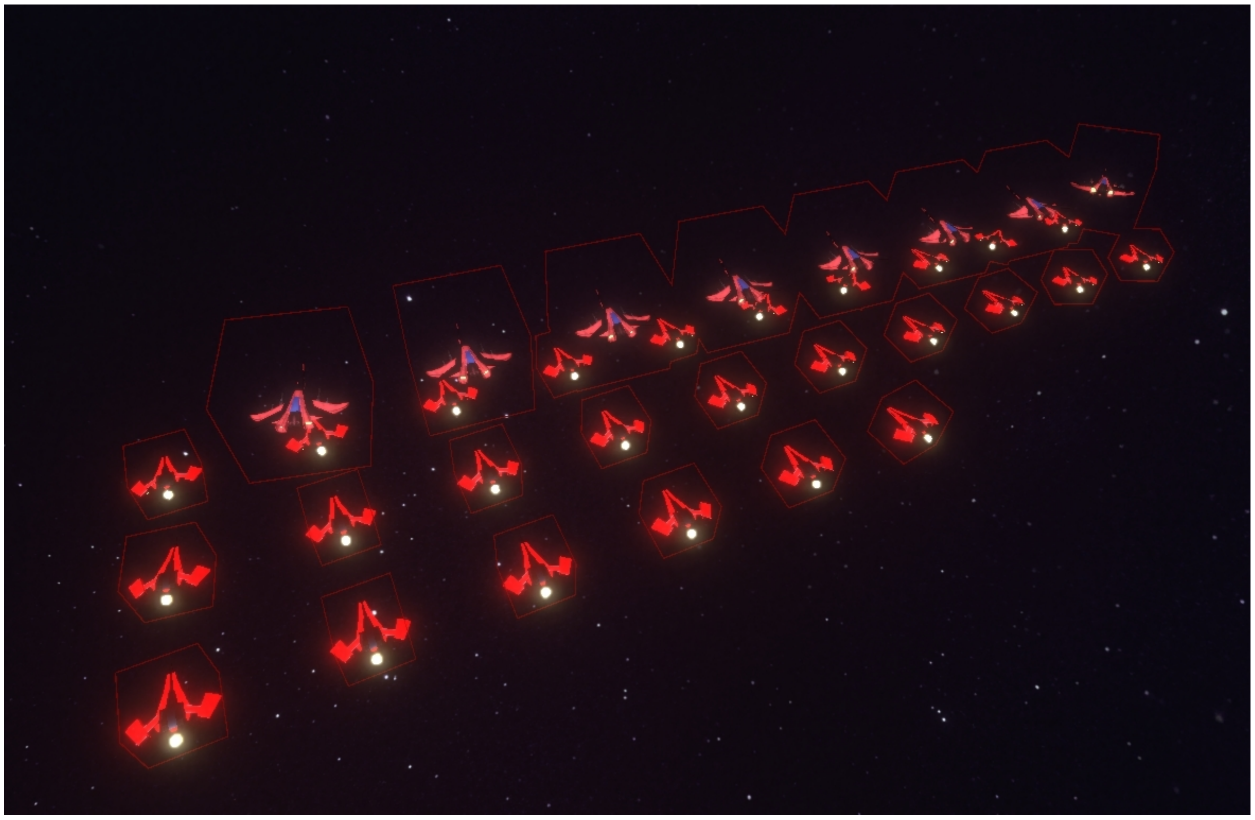


**Figure 19: Units Organized in a Formation**

The player can simply right-click on a position to begin forming a line and then hold and drag the cursor toward the desired position. The length and height of the line adjust according to the distance between the first click and the current position of the mouse. This feature enables players to create a tactical formation that can move together, focus firepower on specific areas and, ultimately, gain an advantage over their opponents. Pseudocode for these features is described in Figure 20.

```
function CreateBoxFormation(unitsByRange, allUnits):
    currUnitsIterated = 0

    // Iterate over each group of units ordered by range
    foreach unitList in unitsByRange ordered by unitList.Key:
        positions = []

        range = unitList.Value[0].range
        distanceBetween = unitList.Value[0].AIPath.maxGroupingDistance
        totalUnits = unitList.Value.Count

        distance = distance(startLinePos, endLinePos)
        if distance < distanceBetween:
            return

        normalizedVec = normalize(endLinePos - startLinePos)
        normalizedRotationVec = crossProduct(normalizedVec, upVector)

        maxUnitsOnARow = totalUnits
        unitsPerRow = distance / distanceBetween
        rows = min(maxUnitsOnARow, unitsPerRow)
        cols = totalUnits / rows

        if totalUnits > (rows * cols):
            cols++

        // How much to adjust our line so the units are in the middle
        rowOffset = (distance / 2) - ((rows * distanceBetween) / 2)
                                   + (distanceBetween / 2)

        placedUnits = 0

        // Iterate over each row and column, find position
        for col from 0 to cols:
            for row from 0 to rows:
                // Calculate the position for this unit
                rowOffsetVec = normalizedVec * (row * distanceBetween)
                colOffsetVec = normalizedRotationVec
                               * ((col * distanceBetween)
                               + (dims * min(distanceBetween, 100)))
                dimOffsetVec = upVector * (dims * min(distanceBetween, 100))
                rowCenteringVec = normalizedVec * rowOffset
                unitLocation = startLinePos + rowOffsetVec
                               + colOffsetVec + dimOffsetVec
                               + rowCenteringVec
                placedUnits++
```

**Figure 20: Pseudocode for Box Formation**

## 6.24 Abilities

RTS games often feature units with special abilities that can give them an edge in combat. These abilities can be used to counter specific enemy units, provide additional support for friendly units, or simply deal extra damage. In addition to providing additional tactical options in combat, special abilities can also help to differentiate units from one another and make each unit feel unique. By using a combination of special abilities and standard attacks, players can develop complex strategies and gain an advantage over their opponents. The following is a list of all the special abilities in the game:

- Speed Boost: The ship's speed is temporarily increased, allowing them to move faster than normal.
- Missile Barrage: The player fires a barrage of missiles at their enemies, causing damage.
- EMP: The player emits an electromagnetic pulse (EMP), disabling electronic devices in the area.
- Shield Boost: The unit's shield regeneration is temporarily increased.
- Laser Barrage: The ship fires a barrage of lasers at its enemies.
- Invisibility: The ship becomes invisible, making it harder for enemies to detect and attack it.
- Kamikaze: The ship charges into an enemy and self-destructs, causing a large amount of damage.
- FTL Inhibitor: The unit activates a special ability that allows other ships to teleport to it.
- Force Field: The unit generates a protective force field in front of it, blocking enemy attacks.
- Obfuscate: The unit has all enemies in range target them instead of anyone else.
- Nuclear Missile: A massive AOE attack that damages all units in a massive radius.
- Reinforcements: Summon smaller units to reinforce carriers or dreadnoughts

To make each special ability have a unique feel, we combined different techniques, including sound effects, particle effects, shaders, and other visual effects. With these techniques, we were able to give players a new layer of control and provide a distinct experience when using each special ability in the game.

## 6.24 Projectile Trajectory Prediction

The following equations below describe the mechanisms behind leading projectiles toward units that have static velocity vectors. We wanted units to be able to predict the next position that a unit would be in, so we could fire at that position.

Given the shooter's position $P_s$ and velocity $V_s$ and target's position $P_t$ and velocity $V_t$, we calculate the relative velocity of the target $V_r$ with respect to the shooter:

$$V_r = V_t - V_s$$

and the relative position

$$P_r = P_t - P_s$$

We then calculate the time $t$ needed for the shot to intercept the target using the function $F$, which returns the intercept time $t$ in seconds. The function takes the shot speed $s$, the relative position $P_r$, and relative velocity $V_r$ respectively.

Finally, we use the intercept time t to find the position where the shot should be fired $P'$.

$$P' = P_t + t * V_r$$

This is the position that the target will be at when the shot arrives, assuming that the target continues to move with the same velocity vector $V_t$ during the time it takes the shot to travel.

We must analyze a few distinct cases:

**Case 1:** if $|V_r|^2 < 0.001$ then we can simply fire regardless of velocity.

We will then calculate $a$, $b$, and $c$ coefficients for the following equation, and factor it to get the intercepts.

$at^2 + bt + c = 0$

Where

$a = |V_r|^2 - s^2$

$b = 2 * V_r \cdot P_r$

$c = |\mathrm{P}_r|^2$

**Case 2:** if $|a| < 0.001$ then $t = -|V_r|^2$ where t must be greater than 0 since we can't shoot back in time.

**Case 3:** Assuming none of the above cases have been met, we then calculate the determinant $D$

$D = b^2 - 4ac$

There are several subcases to account for since there can be one or two intercept points, one in the past and one in the future. There is also a very rare case that both of these are the same, which must also be accounted for.

**Case 3.1:** If the determinant is negative there is no solution, so $t = 0$.

**Case 3.2:** If the determinant is exactly 0, there is one solution.

$t = -\frac{b}{2a}$ assuming $t \geq 0$

**Case 3.3:** If the determinant is positive, we factor the two possible times $t_1$ and $t_2$

$t_1 = \frac{-b+\sqrt{D}}{2a}$ and $t_2 = \frac{-b-\sqrt{D}}{2a}$

We then pick whichever has the shortest, positive time to intercept.

## 6.3 3D Pathfinding

The task of converting a 2D game to a 3D version is a significant challenge, requiring a considerable amount of effort to address without compromising hardware performance. To achieve the desired results, a multi-faceted approach was employed, utilizing a combination of algorithms to ensure sensible unit behavior. This process involved several stages, including the identification of key areas that needed to be addressed to successfully transition the game from 2D to 3D. Another challenge involved ensuring that the game's performance was not adversely affected by the conversion to 3D. This was accomplished by implementing algorithms that were specifically designed to optimize unit behavior, thereby minimizing the amount of processing power required to render the game world.

## 6.31 Octree

An Octree (Figure 21) is a tree data structure used to represent 3-dimensional space by recursively subdividing it into eight smaller regions, or "octants". Each octant is defined by a bounding box that completely encloses a subset of the space, and each level of the tree subdivides the space into increasingly smaller octants. The subdivision process of an Octree starts with a root node that represents the entire 3D space. The root node is then subdivided into eight octants, each representing a smaller portion of the space. If an octant contains too many objects, or if it is too large relative to the objects contained within it, it is further subdivided into eight smaller octants. This process continues recursively until each octant contains either no objects or a small enough number of objects to make further subdivision unnecessary.

This process allows for greater precision when dealing with objects in each scene, while also reducing the memory requirements for areas that are empty. Initially, we considered an approach that involved creating a 3D grid of evenly spaced point nodes and linking them to their nearest neighbors. However, this approach was found to be impractical due to requiring approximately 200,000 nodes for a modestly sized area, which would necessitate an additional memory requirement of around 9 GB. To overcome this challenge, we integrated an API that features an octree into our codebase, which reduced the number of generated nodes to around 18,000, resulting in minimal memory requirements of around 1GB. We designated an object in the scene to contain all the models we wanted to pathfind around and then serialized all the information into storage. By doing so, we optimized the processes even further by pre-calculating the tree's generation.

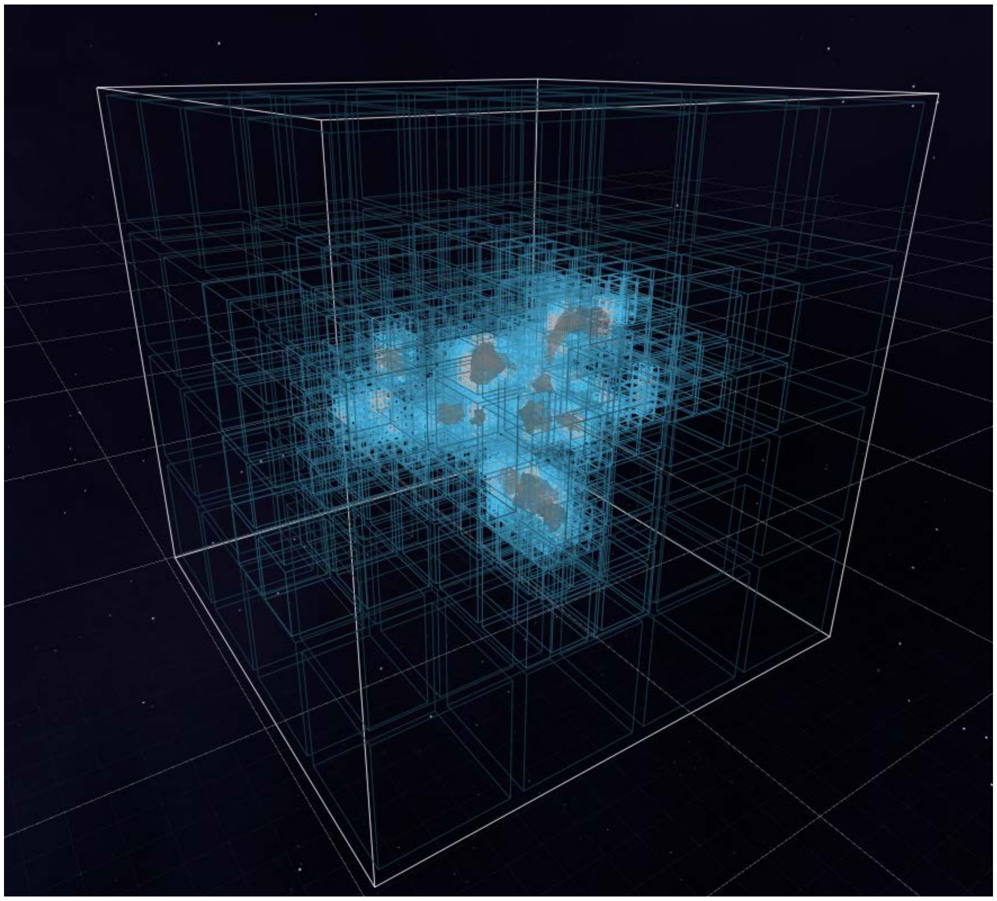


**Figure 21: An Octree Data Structure**

## 6.32 Recursive Linearization

During the development process, we faced a pathfinding challenge that was specifically related to the generation of nodes in the Octree. This issue arose when traversing through large empty spaces, which caused unpredictable behavior in pathfinding due to the scarcity of nodes available. To overcome this challenge, we designed a recursive algorithm (Figure 22) that can accurately calculate the shortest line distance using raycasts, which significantly improves the quality of pathfinding. Interestingly, in cases where no objects are detected between the path, this algorithm eliminated the need for using the A* algorithm altogether.

```
LinearizePath(nonHeuristicPath):
    if nonHeuristicPath.Count <= 2:
        return nonHeuristicPath

    heuristicPath = []

    if not object between first and last point in nonHeuristicPath:
        heuristicPath.Add(nonHeuristicPath[0])
        heuristicPath.Add(nonHeuristicPath[nonHeuristicPath.Count - 1])
        return heuristicPath
    else:
        // Split list into 2 (equal) parts
        firstList = []
        secondList = []
        for i from 0 to nonHeuristicPath.Count - 1:
            if i < (nonHeuristicPath.Count / 2):
                firstList.Add(nonHeuristicPath[i])
            else:
                secondList.Add(nonHeuristicPath[i])

        linearizeFirstList = LinearizePath(firstList)
        linearizeSecondList = LinearizePath(secondList)
        linearizeFirstList.AddRange(linearizeSecondList)
        return linearizeFirstList
```

**Figure 22: Path Linearization Pseudocode**

## 6.33 DOTween

To further enhance the gameplay experience, the development team utilized an API called DOTween, which enabled the implementation of a multi-threaded movement system once the paths were generated, allowing for smoother and more natural movement. The initial list of nodes returned from the A* algorithm is modified using Catmull-Rom splines (Figure 23) to smooth out the paths. Catmull-Rom splines are a type of mathematical curve used in computer graphics and animation that allow for smooth interpolation between points, by considering the values of adjacent points to generate a curve that passes through each of them. This approach helped to eliminate any unnecessary jittering or abrupt changes in direction that could detract from the player's experience.

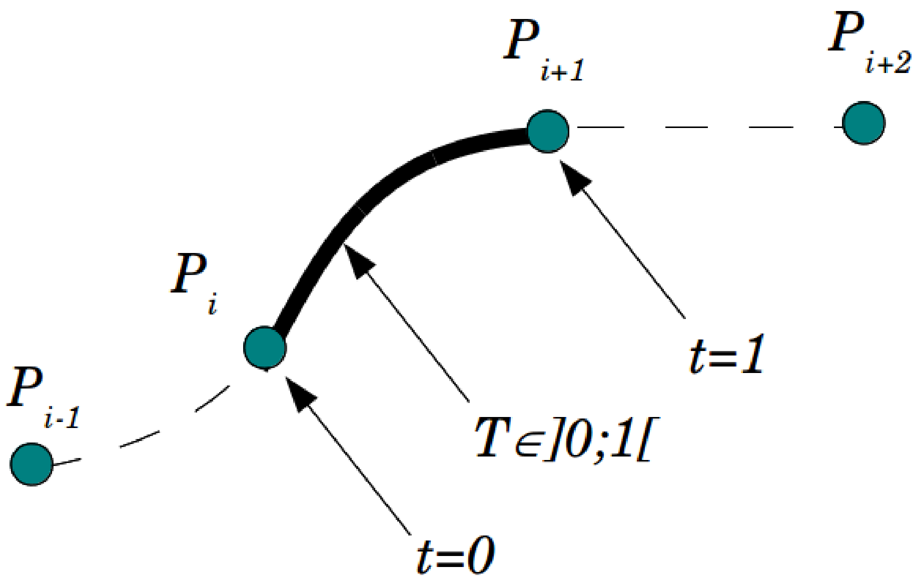


Figure 23: Catmull-Rom Spline

To further improve the flow of movement, each path was divided into three distinct segments: acceleration, constant speed, and deceleration (Pseudocode detailed in Figure 24). We used easing functions, which are mathematical functions used to control the rate of change of an animation or transition, to start and end our movement more smoothly. we were able to accurately determine the required time with derivatives to identify the point where the velocity at the final point of the first list equaled the velocity of the first point of the second list. This ensures a smooth transition from list to list.

```
function SplitAndInterpolate(float firstCutoff, float secondCutoff, float
totalDistance, Vector3[] path) -> (Vector3[], Vector3[], Vector3[]):

    // Create lists to hold the start, middle, and end paths
    startPath = new List<Vector3>()
    middlePath = new List<Vector3>()
    endPath = new List<Vector3>()

    // Split the path into the three sections based on the cutoff distances
    startPath = ListCut(0, firstCutoff, path)
    middlePath = ListCut(firstCutoff, secondCutoff, path)
    endPath = ListCut(secondCutoff, totalDistance, path)

    // Interpolate to find the midpoint vectors
    mid1 = GetMiddleVector(firstCutoff, path.ToList())
    mid2 = GetMiddleVector(secondCutoff, path.ToList())

    // Add the midpoint vectors to the paths
    startPath.Add(mid1)
    middlePath.Insert(0, mid1)
    middlePath.Add(mid2)
    endPath.Insert(0, mid2)

    // Add the final value to the end path
    endPath.Add(path[path.Length - 1])

    // Return the arrays
    return (startPath.ToArray(), middlePath.ToArray(), endPath.ToArray())
```

**Figure 24: Pseudocode to Subdivide a List of Vectors into 3 Separate Lists**

## 6.34 Unit Movement Organization

In an RTS game, it is important to assign positions to units so that they do not bump into each other while moving toward their destination as shown in Figure 25. This can be a complex problem to solve, as there may be multiple units and positions to consider in a large 3D space.

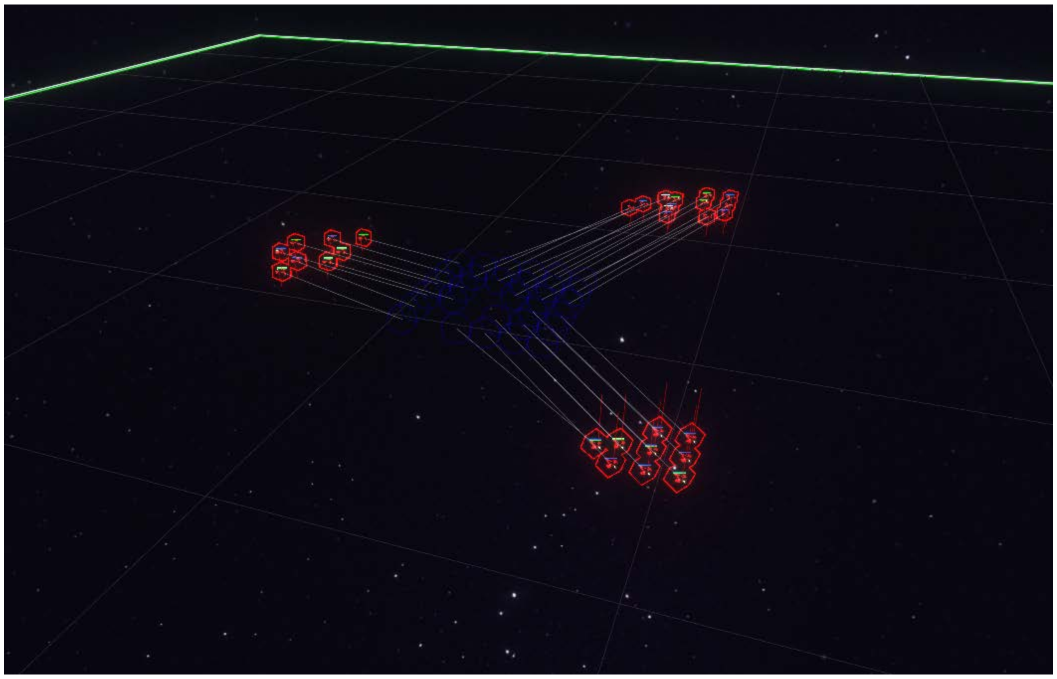


**Figure 25: Three Groups of Units Organizing Themselves Without Colliding**

An algorithm must be employed that can minimize collisions between units attempting to organize themselves into a formation. Specifically, the algorithm must consider the positions of all units and recursively determine the optimal position of each unit in the formation to minimize collisions. The following code (Figure 26) provides a detailed explanation of the algorithmic steps involved in achieving this task.

```
AssignAIPositions(remainingUnits, remainingCoordinates):
    closestUnitPositions = Dictionary<Vector3Int, List<Unit>>()

    for each currUnit in remainingUnits:
        closestDistance = Infinity
        closestCoord = negativeInfinity

        for each currCoord in remainingCoordinates:
            currDistance = Distance(currUnit.position, currCoord)

            if currDistance < closestDistance:
                closestCoord = currCoord
                closestDistance = currDistance

        if closestDistance == Infinity or closestCoord == negativeInfinity:
            throw Exception("Unit could not find a closest position!")

        if closestCoord not in closestUnitPositions:
            unitList = List<Unit>{currUnit}
            closestUnitPositions.Add(closestCoord, unitList)
        else:
            closestUnitPositions[closestCoord].Add(currUnit)

    for each entry in closestUnitPositions:
        farthestPosition = NegativeInfinity
        farthestUnit = null

        for each currUnit in entry.Value:
            currentDistance = Distance(currUnit.position, entry.Key)

            if currentDistance > farthestPosition:
                farthestPosition = currentDistance
                farthestUnit = currUnit

        if farthestPosition == NegativeInfinity or farthestUnit == null:
            throw Exception("Could not find the furthest position!")

        remainingCoordinates.Remove(entry.Key)
        remainingUnits.Remove(farthestUnit)


        farthestUnit.SetDestination(entry.Key)

    if remainingUnits.Count > 0:
        AssignAIPositions(remainingUnits, remainingCoordinates)
```

**Figure 26: Unit Movement Organization Pseudocode**

The pseudocode represents a recursive algorithm for assigning positions to a group of units. For each remaining unit, the algorithm calculates the closest coordinate from a given set of remaining coordinates. If multiple units select the same closest coordinate, the farthest unit from that coordinate is chosen to move to that position. This selection is done iteratively until all remaining units have been assigned a position.

## 6.4 Artificial Intelligence

### 6.41 Reinforcement Learning

A significant goal of our project was to implement a functional and competent Artificial Intelligence (AI) system to play against the player, as is the case with almost all RTS games. We attempted multiple solutions to this problem, from reinforcement learning to state machines.

We aimed to develop a neural network that could replicate the actions of a typical RTS player, specifically, moving friendly units to attack enemies. We trained the model in the Unity Game Engine, utilizing the ML-Agents package which implements PyTorch, a free open-source machine learning framework, to run neural networks. Our initial model had 2 hidden layers, each with 128 neurons. The number of input layers varied based on the number of units simulated, and the output was two floats that determined the X and Z position for the units to move towards.

The goal was for the model to learn the optimal positioning to attack a fleet of enemy ships. The main challenge was determining what inputs to use and the desired complexity of the output. This was because we had access to a wide range of unit statistics constantly updated in real time, the most prevalent of which was a near-infinite amount of 3D coordinates. Since we trained the model in the Unity Engine, we created a facade design pattern to easily access all unit variables and designed a scene to reward and punish the model appropriately. Our reward function provided outputs based on friendly and enemy health, and we aimed to maximize the reward based on the positioning of all units.

We created a cumulative reward function as follows:

$R = r_P \gamma_P - r_N \gamma_N - c$

$R =$ cumulative reward

$r_P =$ positive reward: friendly current health/friendly total health:

$\gamma_P =$ positive reward scaling constant

$r_N =$ negative reward: enemy current health/enemy total health

$\gamma_N =$ negative reward scaling constant

$c =$ constant negative reward

The following is one of our earlier models, we simplified the practically endless 3D coordinates into just using the positions of the enemy units. It also

```
Input Layer (21 units):
  Agent Inputs[5]: {
    %health
    %shield
    targetUnit (action index, not actual unit)
    range
    unitClass
  },
  Enemy Unit 1 Inputs[8]: {
    %health
    %shield
    health ratio
    shield ratio
    targetUnit (action index, not actual unit)
    range ratio
    unitClass ratio
    distance from Agent
  },
  Enemy Unit 2 Inputs[8]: {
    ... (same as Enemy Unit 1)
  },

Hidden Layer 1 (10 units)

Output Layer (2 units): {
  0 -> move towards/target the first enemy unit
  1 -> move towards/target the second enemy unit
}
```

Figure 27: Early Reinforcement Learning Model

The results for this model are as follows, the blue line is an older model that we wanted to improve upon, and the pink line is the model above. As can be seen in Figure 28, the newer model failed to make any significant progress.

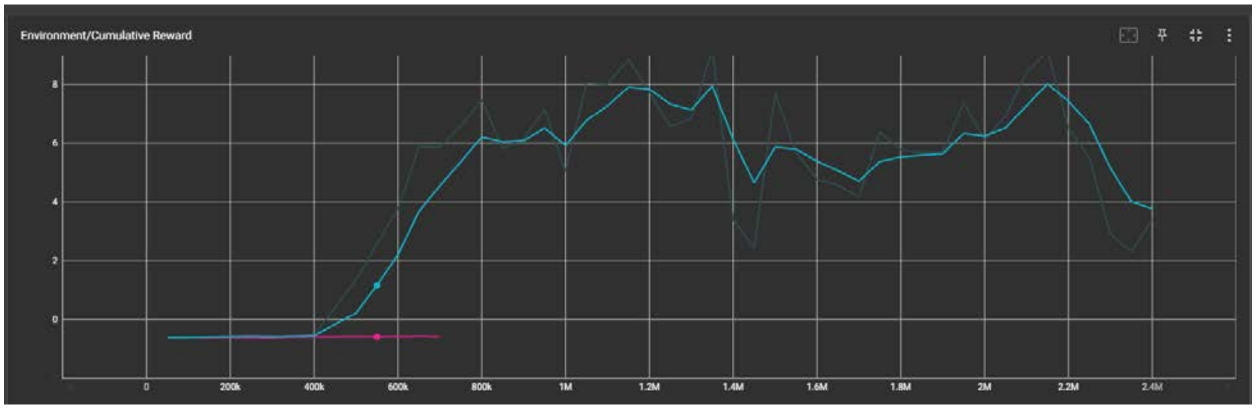


Figure 28: Steps vs. Reward - Newer efficient model failed to improve at all after 700k steps

This trend of more efficient models failing continued. The last model we tested had 2 layers with 128 hidden units, but this had limited success. The time required to train clashed with our time constraints as we were not getting the desired results quickly enough. For example, we ran this model for 90 minutes just for one training session, and it was struggling to move ships to attack other ships consistently. While there was definite progress and improvements in some areas such as larger-scale manipulation with more units, the model struggled with the basics of strategy and small-scale engagements. Also due to the model being trained in Unity the training time was very slow, and making the models more efficient and taking less time was making the results worse.

## 6.42 State Machine

Because we were dissatisfied with the development time of reinforcement learning compared to the results achieved, we are now utilizing a state machine that while less complex, can emulate effective strategy through hard-coded decision-making. The machine has 3 main states that it puts the units into: attacking, defending, and retreating.

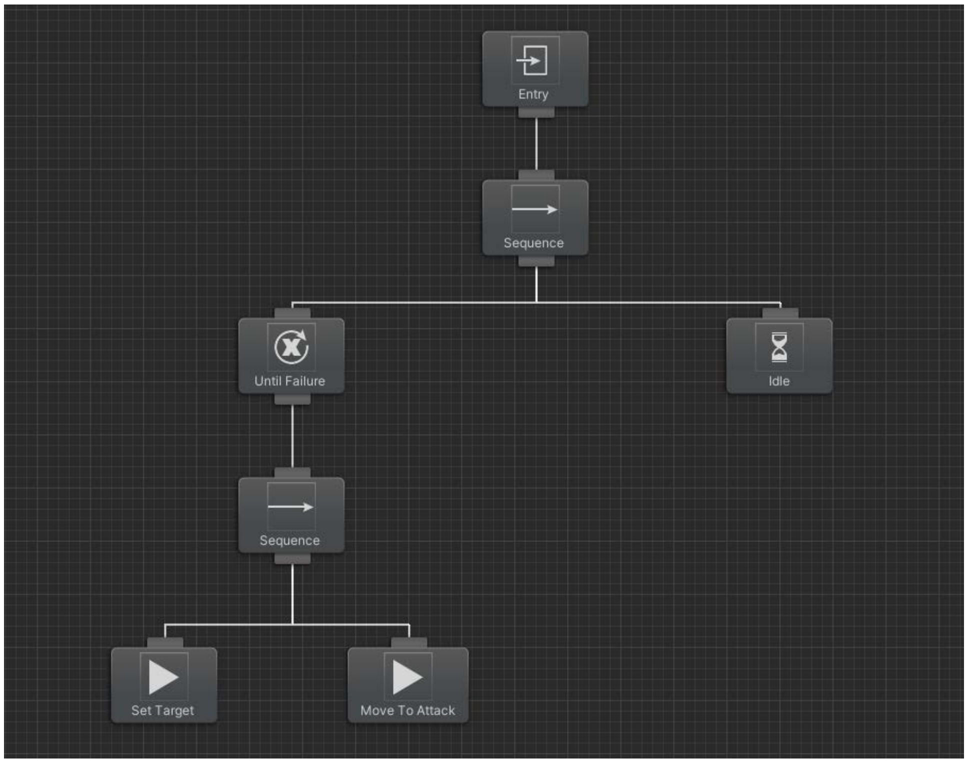


**Figure 29: Flowchart of Current State Machine**

This state machine allows us to control the outcome of the AI's actions more directly, rather than adjusting and hoping things will work after an hour of training. This also allowed for a more expedient and functional deliverable. The state machine essentially gives a list of player units and determines which to attack based on a given utility score. The utility score is how useful the AI determines a ship is to the player and is calculated by the following equation:

*Utility = (Ship Size * 500) - (Current Health + Current Shields) – Distance from Center of Allies/2.5*

Once the utility scores are calculated for each unit, it sends a calculated number of units to attack the players' forces. The higher the utility score, the more vital the unit, and the more ships that will be sent after it. We have begun work to improve the efficiency of this action to prevent sending too many ships or too few. To achieve this, we use a grouping system employed when the player moves ships together. The AI then compares its ships with the player's groups of ships and determines what the ideal number of ships would be to send if possible. Then it begins to target ships individually in the group based on their utility score.

## 6.5 Upgrades

During the initial stages of our project, the team proposed implementing upgrades for different ship types, specifically focusing on upgrading a ship's hull and shields. To accomplish this, several brainstorming sessions were conducted to decide on the variants for shields and hulls. The team aimed to create a variety of unique advantages for each variant to avoid having a single dominant strategy. As a result, a total of six shield variants and nine hull variants were designed.

| *Name* | *Type* | *Effects* |
|---|---|---|
| *Siphon* | Shield | Turn 1% of damage output against enemies into shields |
| *Overcharge* | Shield | For each level, overcharge the available shield amount by 5% |
| *Deflector* | Shield | Reduce projectile damage, increased laser/plasma damage |
| *Overclocked* | Shield | Massively increase shield capacity, increase shield recharge time |
| *Retrofit* | Shield | Convert shield completely into health |
| *Recharge* | Shield | Reduce overall shield capacity, massively reduce shield recharge time |
| *Heat Treatment* | Health | For each level, reduce the incoming damage of plasma-based weapons by 5% |
| *Refractive Plating* | Hull | Incoming lasers have a 1% chance to do no damage |
| *Lead-Alloy Plating* | Hull | Reduces nuclear damage |
| *Polarized Alloy* | Hull | Reduces EMP Damage |
| *Conductive Alloy* | Hull | Reduces special ability cooldown |
| *Alkaline Shell* | Hull | Reduces corrosive damage |
| *Vibration-Resistors* | Hull | Reduces railgun/projectile damage |
| *Neutron Dense Alloy* | Hull | Overall Damage Reduction, but slower turn speed and acceleration |
| *Parasitic Exoplates* | Hull | When damaged, drain 5% of the shield systems to heal 1% of the hull |

**Figure 30: Unit Health and Shield Upgrades**

The figure above showcases our final design which included new passive mechanics for shield variants such as faster recharge time for decreased shields, converting 1% of damage output into shield recharge, and putting all of a ship's shield health into the hull. Hull variants, on the other hand, were primarily designed to mitigate different types of damage.

| *Name* | *Base Damage* | *Damage to Shields* | *Damage to Hull* | *Bonus Damage* |
|---|---|---|---|---|
| *Point-Laser* | Very Low | 100% | 100% | Class 2- |
| *Laser* | Medium | 100% | 70% | None |
| *EMP Laser* | Laser Base | 200% | 20% | None |
| *Missile* | High | 10% | 100% | Class 7+ |
| *EMP Missile* | Missile Base | 200% | 10% | Class 7+ |
| *Corrosion Missile* | Missile Base | 0% | 200% | Class 7+ |
| *Railgun* | High | 0% | 100% | Class 7+ |
| *EMP Railgun* | Railgun Base | 100% | 50% | |
| *Plasma* | Medium | 70% | 100% | |
| *Nuclear* | Very High | 100% | 200% | Class 7-8 |

**Figure 31: Table of Weapons and Variants**

To support these changes, a redesign of the projectiles and damage system was necessary since they did not previously have specific types. The team designed seven weapon types with a total of four variants divided among the weapons. In the above figure, any weapon that uses another weapon's base damage is considered a variant, as it is a modification to that weapon. Each weapon dealt varying amounts of damage to shields and had increased damage output to ships based on their size class. Furthermore, two main weapon variants were developed, corrosive and EMP. Corrosive weapons inflicted more damage to the hull while EMP weapons dealt increased damage to shields. However, both variants had the drawback of dealing less damage to the opposing type of ship health. We also originally included in our design a mine weapon that had both a corrosive and EMP variant however due to time constraints we did not end up implementing them.

## 6.6 Veterancy

To encourage players to prioritize strategy and keep their ships alive for as long as possible, we implemented a system known as Veterancy. The Veterancy system is based on an experience point (XP) system, in which each unit has its own XP counter that increases as it participates in battles and survives encounters with enemy ships. The XP counter is further increased if the unit has sustained damage, and the amount of XP a unit earns upon destroying another unit is determined by its size class. Each level gained through Veterancy grants the unit a cumulative increase in total health, shields, and damage output. The equation of which can be seen below:

$$Value\ Percent\ Increase = 1 + (Veterancy\ Level\ / \ 10)$$

To visually signify this progression, we added particle effects and UI elements in the game to show when a unit has leveled up and its corresponding rank. The amount of experience points required to reach each level is universal amongst all units, after which their points are set to 0. The leveling curve is as follows:

| *Level* | *XP* |
|---|---|
| 1 | 100.00 |
| 2 | 150.00 |
| 3 | 225.00 |
| 4 | 337.50 |
| 5 | 506.25 |
| 6 | 759.38 |
| 7 | 1139.06 |
| 8 | 1708.59 |
| 9 | 2562.89 |
| 10 | 3844.34 |
| 11 | 5766.50 |
| 12 | 8649.76 |
| 13 | 12974.63 |
| 14 | 19461.95 |
| 15 | 29192.93 |
| … | … |
| $a_n$ | $a_{n-1} \cdot 1.5$ |

**Figure 32: Veterancy Values and Equation**

The size of a ship played a significant role in the design of the Veterancy mechanic. As a general rule, larger ships are more effective at destroying other ships, and therefore, it was necessary to ensure that they did not have an unfair advantage in terms of earning experience points. To address this issue, we developed a formula for calculating experience points that took into account the size of the destroyed ship relative to the size of the ship doing the destruction.

$$Actual\ Experience\ =\ Incoming\ Experience\ *\ (Destroyed\ Ship\ Size\ /\ This\ Ship's\ Size)$$

By using this formula, smaller ships that participated in destroying larger ships were rewarded with more experience points than they would have earned from destroying smaller ships. This approach introduced a risk versus reward element to combat and encouraged players to make tactical decisions that prioritized the survival of experienced ships.

Overall, the implementation of this formula ensured that players were incentivized to engage in strategic combat that considered the size of their ships and the ships they were facing, rather than simply farming smaller ships for easy XP.

## 6.7 Structures

Imperius has a sophisticated economic system that enhances the gameplay experience by introducing resource management elements. The game features three main resources that players must manage: Power, Supplies, and Tech. Supplies represent the primary resource in Imperius, which players need to build structures, units, and research new technologies. Power serves as a limiting agent to prevent players from mass-producing supply structures early in the game. As players build more structures, their power consumption increases, requiring the construction of additional power structures to maintain a balance. If players consume too much power, they won't be able to create new buildings and their supply rate will decrease substantially. Construction statistics for Imperius ships, divinity ships, and buildings are detailed in Figure , Figure , and Figure  respectively. The main base (Figure ) allows for the construction of buildings.

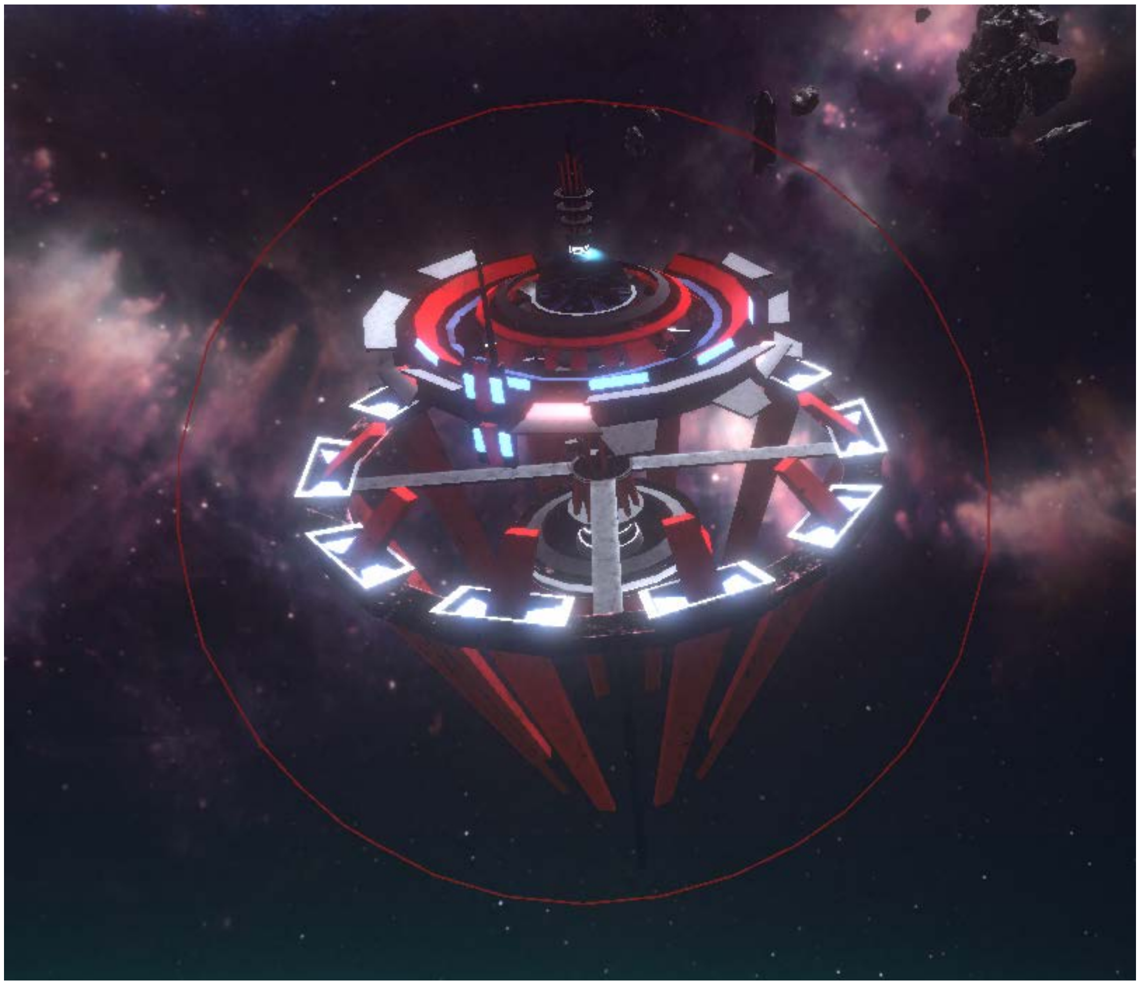


Figure 33: The Main Base of the Imperius Faction

Upgrading to higher tech levels unlocks new buildings, ships, and upgrades, providing players with a wide range of strategic options to choose from. Three buildings contribute to unit production: the main base, dockyard, and shipyard. The main base can only produce fighters, while the dockyard and shipyard are responsible for smaller and larger ships, respectively.

| *Building Name* | *Function* | *Cost* | *Tech Level* | *Build Time (seconds)* |
|---|---|---|---|---|
| *Command Center* | Main | 10000 | 4 | 120 |
| *Temple* | Main | 10000 | 4 | 120 |
| *Solar Array* | Power | 2500 | 0 | 30 |
| *Shrine* | Power | 2500 | 0 | 30 |
| *Dockyard* | Unit Production | 4500 | 2 | 45 |
| *Monastery* | Unit Production | 4500 | 2 | 45 |
| *Shipyard* | Unit Production | 7500 | 3 | 60 |
| *Cathedral* | Unit Production | 7500 | 3 | 60 |
| *Logistics Hub* | Supply | 2000 | 0 | 30 |
| *Requisitioner* | Supply | 2000 | 0 | 30 |
| *Defense Platform* | Defense | 2500 | 1 | 30 |
| *Sacred Flame* | Defense | 2500 | 1 | 30 |

**Figure 34: Building Construction Parameters**

In Imperius, turrets serve as essential stationary defenses against enemy attacks, providing players with an additional layer of defense to protect their bases and resources from enemy assaults. To succeed in Imperius, players must effectively manage their resources, build the right mix of units and structures, and deploy their defenses strategically to fend off enemy attacks. The game also features buildings with hardpoints, like units, which enable more strategic play by allowing players to attack from specific angles to avoid enemy fire.

| *Ship Name* | *Ship Class* | *Tech Level* | *Cost* | *Build Time (seconds)* |
|---|---|---|---|---|
| *Espion* | PatrolCraft | 0 | 500 | 5 |
| *Nerva* | Corvette | 1 | 1000 | 10 |
| *Elysium* | Frigate | 2 | 2000 | 20 |
| *Lodestar* | Destroyer | 2 | 2500 | 60 |
| *Sentinal* | Destroyer | 2 | 2000 | 20 |
| *Noctus* | Destroyer | 3 | 3500 | 30 |
| *Garuda* | Carrier | 4 | 5000 | 45 |
| *Calibre* | Cruiser | 3 | 3500 | 30 |
| *Verdancy* | Cruiser | 4 | 4000 | 45 |
| *Aurelius* | Battleship | 5 | 6000 | 60 |
| *Solace* | Battleship | 5 | 6000 | 60 |
| *Pacific* | Dreadnought | 5 | 50000 | 120 |

**Figure 35: Build Parameters of Imperius Units**

| *Ship Name* | *Ship Class* | *Tech Level* | *Cost* | *Build Time (seconds)* |
|---|---|---|---|---|
| *Argo* | PatrolCraft | 0 | 500 | 5 |
| *Nile* | Corvette | 1 | 1000 | 10 |
| *Ishim* | Frigate | 2 | 2000 | 20 |
| *Sladen* | Destroyer | 2 | 2500 | 25 |
| *Censeur* | Destroyer | 2 | 2500 | 25 |
| *Zaraton* | Carrier | 3 | 3000 | 30 |
| *Castilian* | Cruiser | 3 | 3500 | 45 |
| *Taniwa* | Cruiser | 4 | 4000 | 45 |
| *Aeneas* | Cruiser | 4 | 4000 | 45 |
| *Seraphim* | Battleship | 5 | 5000 | 60 |
| *Polyanthus* | Battleship | 5 | 10000 | 80 |
| *Akuma* | Dreadnought | 5 | 50000 | 120 |

**Figure 36: Build Parameters of Divinity Units**

## 6.8 Multiplayer

During the final stages of our project, we decided to incorporate networking capabilities, (Figure ) which presented significant challenges. We utilized Unity's Netcode for GameObjects, a relatively new networking solution that simplified the process considerably. Additionally, we used Facepunch Transport to connect to Steam, which presented unique challenges of its own.

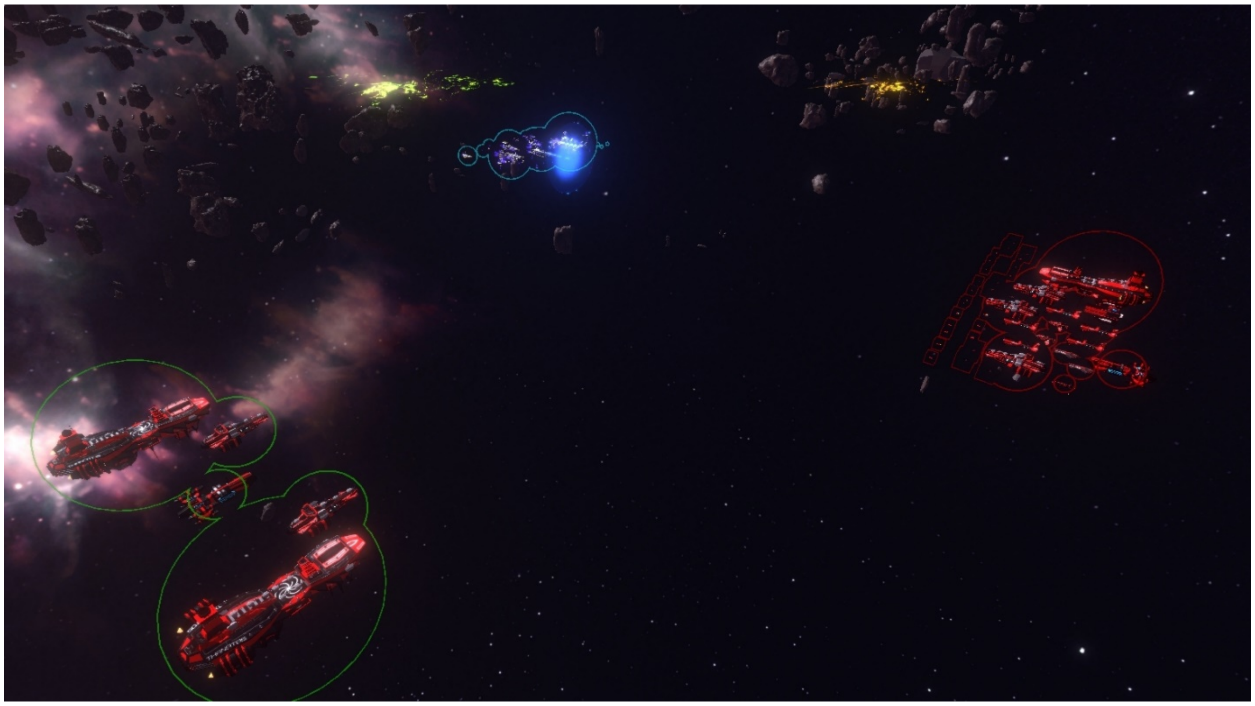


**Figure 37: Three Players in a Game**

Unity's networking solution employs a protocol that is similar to UDP (User Datagram Protocol) to transmit data between clients and servers. UDP is a lightweight protocol that is commonly used in real-time applications such as video streaming and online gaming. However, Unity's implementation deviates from traditional UDP by incorporating some reliability and sequencing features to ensure that important data is transmitted properly while still maintaining the low latency and responsiveness required for real-time applications.

## 6.81 Remote Procedure Calls and Network Variables

```
public void SetDestination(Vector3 position)
{
    if (canMove)
    {
        if (IsHost)
        {
            SetDestinationClientRpc(position.x, position.y, position.z);
        }
        else
        {
            SetDestinationServerRpc(position.x, position.y, position.z);
        }
    }
}

private void SetDestinationInternal(float3 position)
{
    if (isHero && heroDown) return;
    AIPath.SetPath(position);
    AIPath.RemoveChain();
}

[ServerRpc(RequireOwnership = false)]
private void SetDestinationServerRpc(float x, float y, float z)
{
    SetDestinationClientRpc(x, y, z);
}

[ClientRpc]
private void SetDestinationClientRpc(float x, float y, float z)
{
    SetDestinationInternal(new Vector3(x, y, z));
}
```

**Figure 38: Code for Setting a Destination with RPCs**

One of the main methods used in implementing networking in our game was using Remote Procedure Calls (RPCs) to send data between clients and servers. RPCs allow a client to call a function on the server and receive a response as if the function was running locally on their machine. This is necessary for authoritative server architecture, where the server has complete control over the game state, and clients cannot make any changes without the server's approval. There are two main types of RPC: Server and Client. ServerRPCs are sent from clients to the server and authenticated appropriately, sending information that the server needs to know about the client’s wishes to change the game state. ClientRPCs have the opposite effect, they send data to the server from the clients to keep the game state updated. We can use a combination of server and client

RPCs to have clients request changes to the game state, and then forward those changes to the other clients. Figure  details code for a basic implementation of both Client and Server RPCs. When we need to move somewhere, we call the SetDestination() method. Since the host can only call ClientRPCs, a client requesting to move a unit must first send ServerRPC to request to move said unit. Once received, the server creates a ClientRPC, which tells all relevant clients to move a unit to the specified position.

Server authoritative architecture is essential for security reasons as it prevents cheating and other malicious behavior. It also allows for an easy way to implement both AI players and observers in the game since the server has control over the entire game state for all players. We simply need to remove or add permissions to specific players depending on what the server decides their assigned role is. Server authoritative architecture also simplifies the process of modifying the codebase, as we can simply find specific methods that control vital functions of the game state (such as setting the destination of a unit) and turn them into Client and Server RPCs. Thus, RPCs constitute the primary data traffic in the system, serving as fundamental requirements for the server authoritative architecture. In compliance with this architecture, every client-initiated object movement is authenticated through an RPC, and the server subsequently sends an RPC to inform the client about the updated game state reflecting their decision.

Network variables also play a critical role in transmitting data from server to clients, such as the health and shield values of individual units. Network variables allow an easy way to serialize and send data over a network with certain permissions and specific owners. For our purposes, all this data was written by the server and read by the clients.

## 6.82 Network Resilience: Packet Loss and Latency Differences

To ensure that the game state remained synchronized amongst players even in high latency and packet loss situations, we employed several techniques. We calculated the average time to transmit information between servers and clients and only displayed certain events after compensating for this delay.

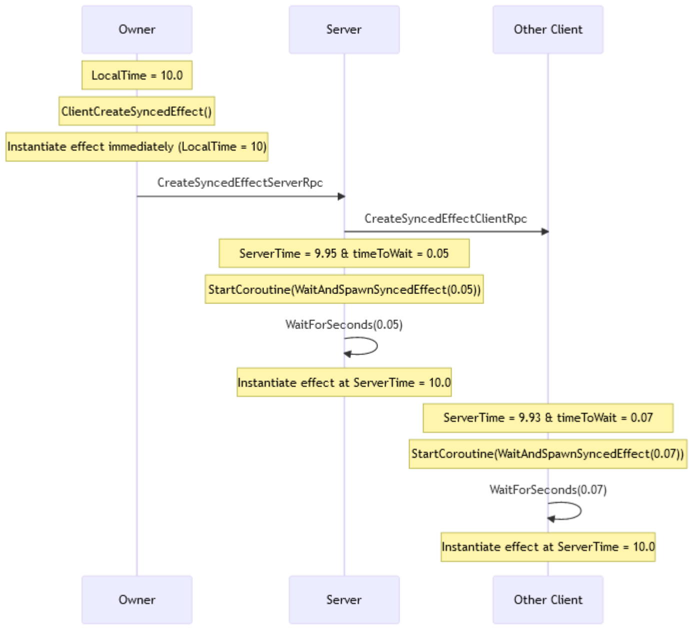
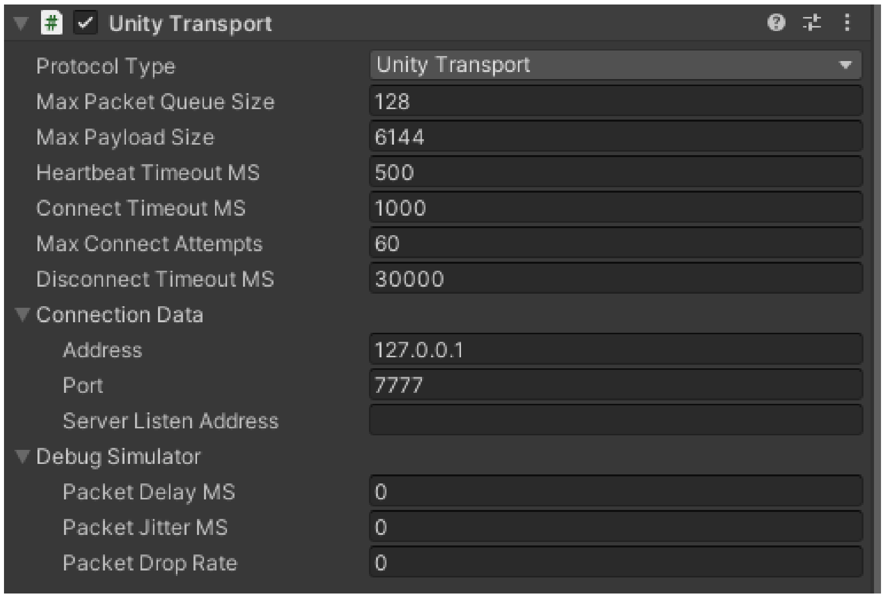

Owner
Server
Other Client
LocalTime = 10.0
ClientCreateSyncedEffect()
Instantiate effect immediately (LocalTime = 10)
CreateSyncedEffectServerRpc
CreateSyncedEffectClientRpc
ServerTime = 9.95 & timeToWait = 0.05
StartCoroutine(WaitAndSpawnSyncedEffect(0.05))
WaitForSeconds(0.05)
Instantiate effect at ServerTime = 10.0
ServerTime = 9.93 & timeToWait = 0.07
StartCoroutine(WaitAndSpawnSyncedEffect(0.07))
WaitForSeconds(0.07)
Instantiate effect at ServerTime = 10.0
Owner
Server
Other Client

**Figure 39: Latency Compensation Diagram (Credit Unity Technologies)**

Figure  details the delay compensation method used to synchronize events on both client and server. Netcode for GameObjects uses a star topology, where all communications occur between clients and the server/host, and never between clients directly. NetworkTime is used to consider transmission delays that may occur while communicating. There are two different time values: LocalTime and ServerTime. LocalTime on a client is ahead of the server, while ServerTime on clients is behind the server. Network time can be used to ensure that events are triggered precisely on time for all clients, accounting for network transmission delays. When a client wants to trigger an event, it can pass to the server the local time at which the event should be triggered to the server. The server can then subtract its current ServerTime from this value to calculate the amount of time it needs to wait before triggering the event. This delay can be passed back to the client, which can then wait for that amount of time before triggering the event locally.

Similarly, when the server wants to trigger an event on the clients, it can pass the server time at which the event should be triggered to each client. The client can then subtract its current ServerTime from this value to calculate the amount of time it needs to wait before triggering the event. The client can then wait for that amount of time before triggering the event locally. By using network time to calculate the delay, events can be triggered precisely on time for all clients, even those with bad network connections.

In addition, the server authoritative design of the code ensures that any client experiencing desynchronization is automatically resynchronized with the next successful game state update. Critical game parameters, such as unit positions, health values, and current targets, are transmitted regularly from the server to all clients. These values represent the server's current state and are not relative to previously known values, but rather the actual state of the game, which is then interpolated by the clients to ensure a seamless transition between past and present game states. To achieve this shared and robust game state, Unity employs a combination of UDP and TCP protocols. UDP is utilized for unit transforms, such as position, rotation, and scale, where a dropped packet would have a minimal impact given the high volume of information transmitted per second. In contrast, TCP is used in areas where precise synchronization was necessary, such as movement orders or activating special abilities, and implemented via Server and Client RPCs.



Unity Transport

| | |
|---|---|
| Protocol Type | Unity Transport |
| Max Packet Queue Size | 128 |
| Max Payload Size | 6144 |
| Heartbeat Timeout MS | 500 |
| Connect Timeout MS | 1000 |
| Max Connect Attempts | 60 |
| Disconnect Timeout MS | 30000 |
| Connection Data | |
| Address | 127.0.0.1 |
| Port | 7777 |
| Server Listen Address | |
| Debug Simulator | |
| Packet Delay MS | 0 |
| Packet Jitter MS | 0 |
| Packet Drop Rate | 0 |

**Figure 40: Unity Network Simulation Settings**

We were able to use Unity's multiplayer tools package (Figure ) for recording network statistics and the Unity Transport for simulating packet delay, jitter, drop rate, and more as shown in the bottom of Figure . The game's core functionalities and mechanics are maintained seamlessly with network packet loss of up to 20% according to our stress tests. (Figure ) Beyond this threshold, gameplay disruption may arise, leading to the desynchronization of the game state. The underlying cause can likely be traced back to TCP handshake timeouts and other errors triggered by a high volume of packet loss. A possible solution to this issue involves incorporating a TCP-like subsystem without these timeouts that periodically resynchronizes the entire game state at predetermined intervals. However, it is unlikely for packet loss to reach such catastrophic levels of failure given that average network packet loss is usually under 2% in general.

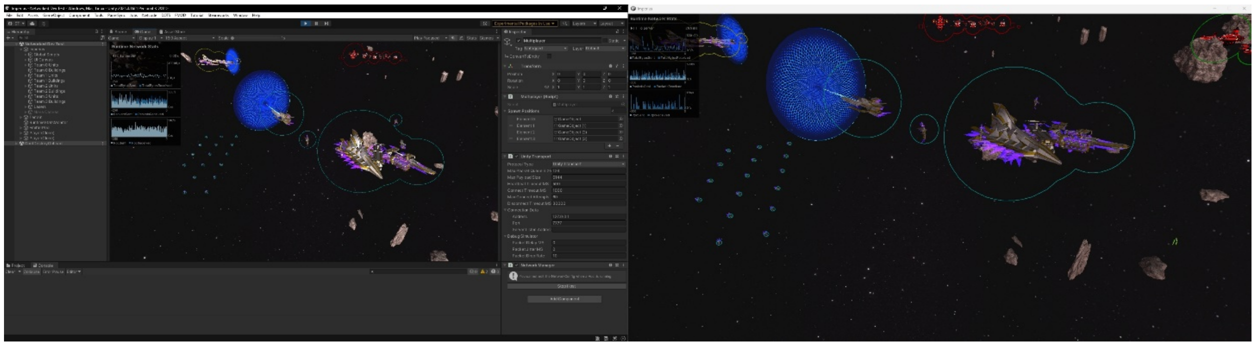


**Figure 41: Synced Game Instances Under Simulated Packet Loss**

Packet delays (ping) also had a significant effect on gameplay. Delays from 0 to 100 ms showed little/no disruption in gameplay, while 100-200 ms showed moderate disruption, and anything beyond significantly affected the gameplay experience or was considered unplayable. All data was obtained through testing with other players via VPN or through simulated conditions.

## 6.83 Network Analysis

As anticipated, the host experiences considerably greater strain than the clients connected to it. The data reveals that the networking metrics of the clients remained relatively consistent across different player counts. However, the host's transmission load in terms of packets and bytes showed a significant increase in direct proportion to the number of players present. Specifically, the server's outbound packet rate was observed to increase by approximately 200 packets per second for each additional player, while the amount of data transmitted also scaled linearly with a rate of approximately 100 bytes per second per player. There were negligible differences between players with different latencies to the server when tested via VPN and other players located in different states. All data beyond 4 players is extrapolated as seen in Figure 42.

| **# of Players** | **RTT to clients (ms)** | **Bytes Sent (kB/s)** | **Packets Sent (#/s)** | **Rpc Sent (k/s)** |
|---|---|---|---|---|
| 2 | 25-50 | 50-150 | 100-200 | 6-8 |
| 3 | 25-50 | 150-250 | 200-400 | 10-12 |
| 4 | 25-50 | 250-350 | 400-600 | 12-14 |
| 5 | 25-50 | 350-450 | 600-800 | 14-16 |
| 6 | 25-50 | 450-550 | 800-1000 | 16-18 |
| 7 | 25-50 | 550-650 | 1000-1200 | 18-20 |

**Figure 42: Server Information Sent to Clients**

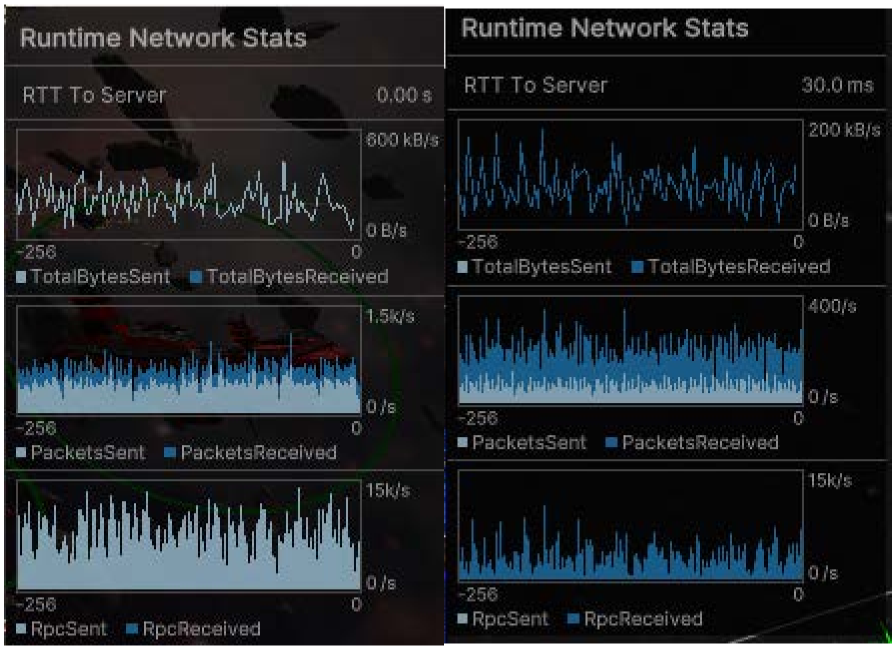


**Figure 43: 4 Player Network Stats (left is server, right is client)**

It should also be noted that there is considerable potential for a distributed system implementation. Since there is an authoritative server implementation, all particle collision messages are handled by the server, and their relevant data is transmitted over the network to each non-host client. Therefore, we can remove a significant portion of CPU calculations normally required from each non-host client, freeing up CPU power for other tasks. If we were to break server-authoritative architecture, we could theoretically share this load across all clients equally or equitably based on hardware to create a faster experience for all players. Since server authoritative architecture was a requirement, the former was implemented instead of the latter.

# 7. Art Design

## 7.1 Direction

The primary goal of the art direction in this project was to create an improved depiction of the original prototype's style. This entailed developing new ship designs, as well as creating and texturing high-quality assets. Throughout the creative process, it was important to ensure that we remained consistent with Imperius' visual identity. As a result, we relied heavily on the former models as a point of reference while we generated new assets. In contrast, the Divinity did not have a set art direction, which provided an opportunity to experiment with different art styles.

Upon examining the relationship between Imperius and Divinity, it becomes evident that Imperius is the true antagonist in the story. To convey this theme, we attempted to emulate a gothic sci-fi art style that featured free-flowing structures with ragged and sharp edges. We found that this particular architectural style evokes a sense of menace and dread. Additionally, we aimed to make Imperius appear industrial and man-made while also emphasizing their emphasis on battle performance over aesthetics.

## 7.2 3D Modeling

To develop the models for the game, the decision was made to utilize Autodesk Maya, a highly sophisticated modeling software widely adopted by professionals within the industry. The team's proficiency in Maya was our strong point, but this was also our first time creating anything of this scale in a genre of art that most of us had never explored before. This meant that although we knew how to use Maya to a certain extent, our knowledge did not extend to modeling something that would be hundreds and thousands of units large. This proved to be a problem that we would begin understanding as we created more and more models, which meant that some of the later models had notably better designs and modeling practices.

### 7.21 Ships

The original ships in the game were created by mirroring one cube, connecting the cube's faces, and then using a series of extrusions for details. These ships were then given basic textures of grey, black, and red, with red being the dominant color for the Imperius ships. As seen in Figure 44, these ships were able to stand on their own, but they also left much to be desired.

To improve upon their designs, we first wanted to stray away from the symmetrical feel that the current designs had. We felt that while most forms of transport are symmetrical, it meant that the Imperius ships felt too generic. Furthermore, we wanted the ships to look less aerodynamic. We felt this would not only make the ships feel more unique but also help tell the players that the ships in this game were only built for one thing: deep-space combat and travel.

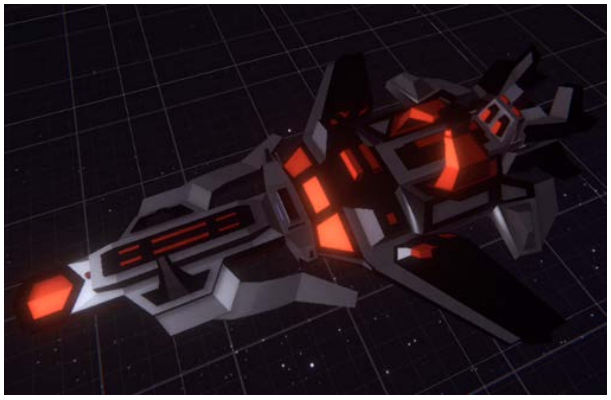


**Figure 44: An Old Imperius Ship Model**

### *7.211 Designing a Ship*

The initial obstacle we faced was the need to create distinct and innovative designs for each spaceship. To overcome this challenge, we undertook extensive research into the various components of spaceships. This involved a thorough understanding of the placement of essential elements such as weapon systems on the deck, the location of the captain's operating room, and minor components like sleeping quarters. To inform our design process, we conducted an in-depth examination of a variety of sci-fi media. We analyzed and took inspiration from different games and movies to identify unique and intriguing ship designs. We then compiled these designs into an artboard, which served as a constant reference throughout the development process. This approach enabled us to create designs that were both original and visually compelling.

### *7.212 Additional Inspiration*

Including the games and franchises we took inspiration from, we also used websites such as ArtStation and Pinterest to further diversify our design ideas. Once we found an interesting ship design, such as those shown in Figure 45, we would then try and replicate these designs into our models, while ensuring that we do not completely mimic the original artwork. This would often vary from simple engine designs to the overall visual balance of a ship. The process of mixing and matching a multitude of designs into each ship meant that our ship designs felt unique yet recognizable. This process of development also helped alleviate the workload which allowed us to create twenty-four unique ships for the Imperius and Divinity.

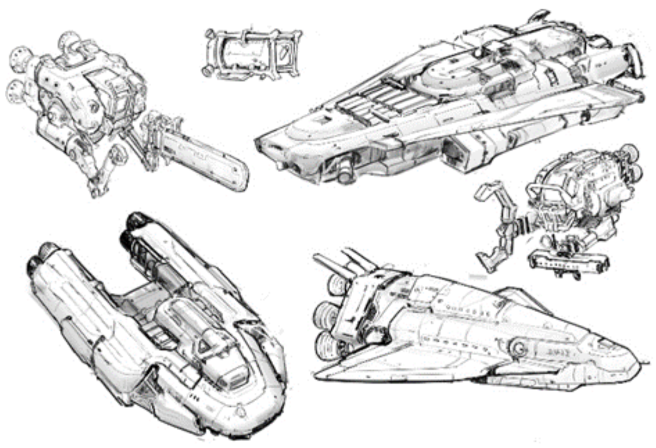


Figure 45: Spaceship Concept Art from ArtStation

### 7.213 Silhouette Test

The utilization of the silhouette test has become a widespread practice in character design and concept art to ensure that each piece of art is distinct and easily recognizable. In our project's developmental stages, we decided to implement this test for each ship design. To do so, we opened each model in 3D Viewer, a user-friendly model viewing application offered by Windows. Once we removed all lighting from the scene, we captured screenshots of the model's silhouette, such as the one shown in Figure 46. These screenshots were then compiled into a comprehensive list for each ship. In instances where we identified two ships that were not readily distinguishable from one another, we would either redesign one of the ships or start again with both ships.

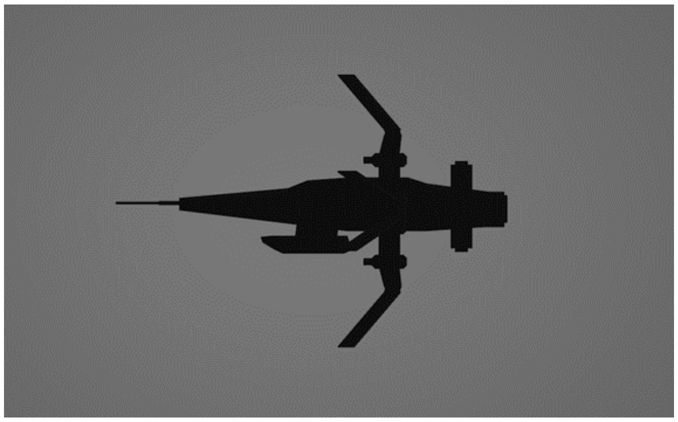


Figure 46: An Imperius Ship Silhouette Viewed in 3D Viewer

### 7.214 Ship Abilities

The ships of the Imperius and Divinity factions boast distinctive capabilities that have significantly influenced their design. For instance, the Imperius Lodestar is equipped with the Faster Than Light (FTL) ability, enabling it to teleport other units to its location. Given its pivotal role in combat scenarios, it was crucial to devise a design that would set it apart from other ships. As seen in Figure , we settled on a final version that featured two towering structures that would glow in-game when its ability was activated. By leveraging the unique capabilities of each ship, our team was able to experiment with novel and innovative designs that defied convention.

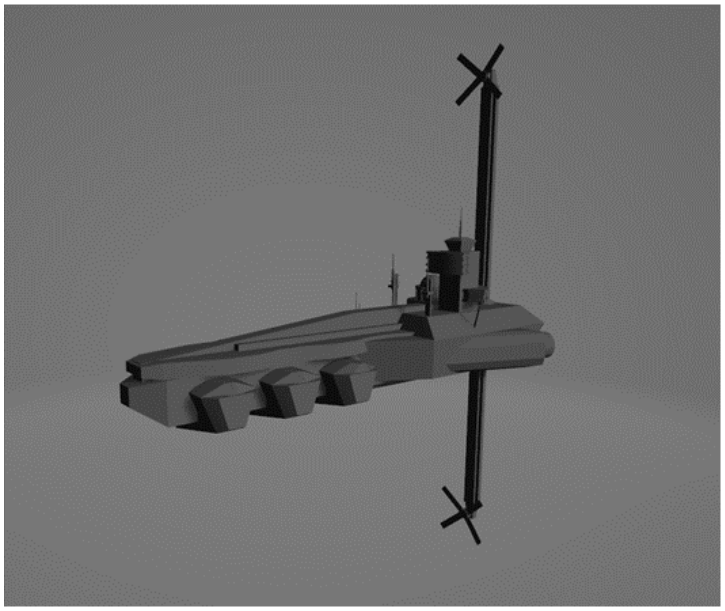


Figure 47: The Imperius Lodestar Ship

### 7.215 First Stage

The first ship we developed was the Imperius scout unit, commonly referred to as the Espion. In the hierarchy of the Imperius fleet, this ship holds the lowest rank both in terms of size and battle power. The primary purpose of creating this ship was to establish a reference point for other ship designs and to serve as an initial platform for testing various modeling techniques.

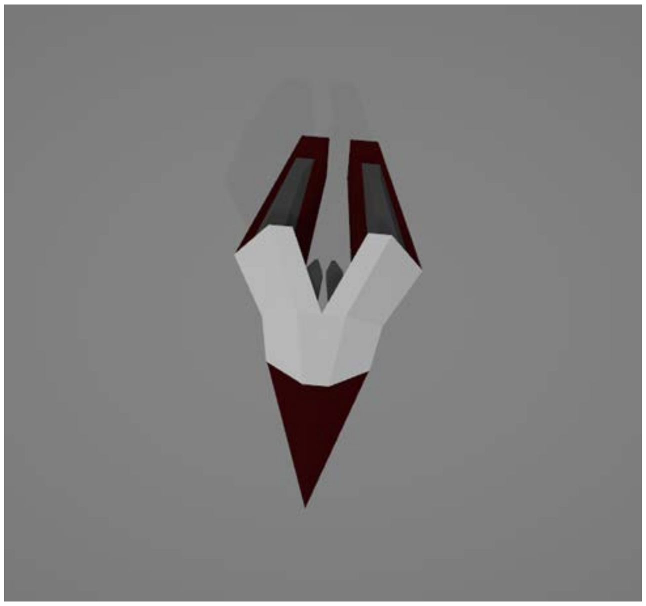


Figure 48: The First Prototype for the Espion Scout

The design concept behind the Espion was to create a ship that embodied the appearance of an arrowhead, similar to what is commonly seen in many games that involve scouting units. These units are intended to function as expendable entities that rapidly scout ahead with minimal armor. To aid in conveying a sense of speed to the player, we decided to shape the Fighter with an aerodynamic profile as seen in Figure . During the development process, we encountered a challenge in determining whether to utilize single or multiple cubes to construct the ship's body. This issue emerged as we continued to refine the Espion's design into its final design, shown in Figure .

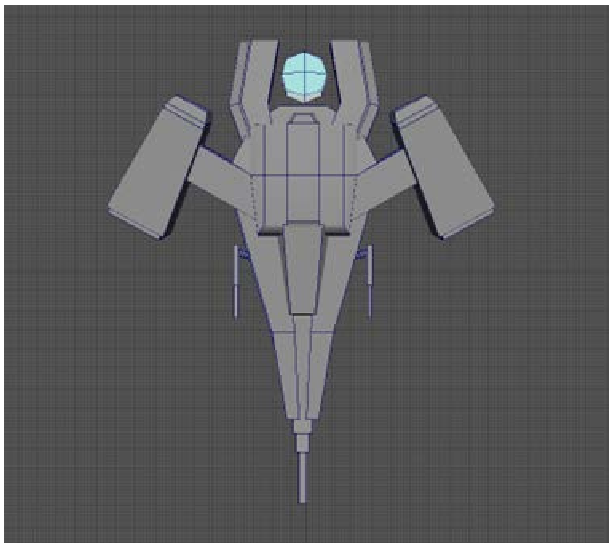


**Figure 49: The Final Reiteration of the Imperius Espion**

## *7.217 Divinity Ships*

Due to time constraints, the modeling of the Divinity ships was conducted using single objects. Although this approach resulted in a decrease in the quality of the ships, we ascertained that this would not pose a significant issue since the Divinity ships would receive the least amount of screen time. To compensate for the reduced level of detail, the ships were fashioned with more jagged shapes and edges, as well as abstract hull shapes as seen in Figure . This design choice provided a stark contrast to the industrial aesthetic of the Imperius ships and allowed for the exploration of more archaic forms and structures.

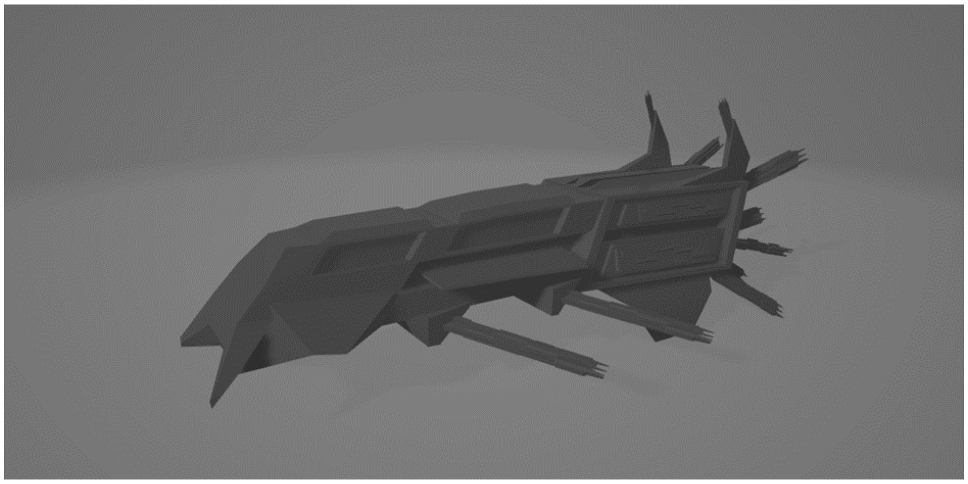


**Figure 50: The Ishim, a Divinity Frigate Ship**

In addition to their unique hull shapes, the Divinity ships were characterized by eccentric spikes protruding from their main body. These spikes served to represent the power gifted to the Divinity and aided in the propulsion of their ships. The initial experimentation with this power occurred in the smaller ships, as evidenced by their rough integration. As the Divinity gained a greater understanding of this technology, they acquired the knowledge necessary to incorporate the spikes into the hulls of their larger ships. This design feature allowed us to explore the limits of extrusion and symmetry, as well as to incorporate verticality into the ship design as seen in Figure .

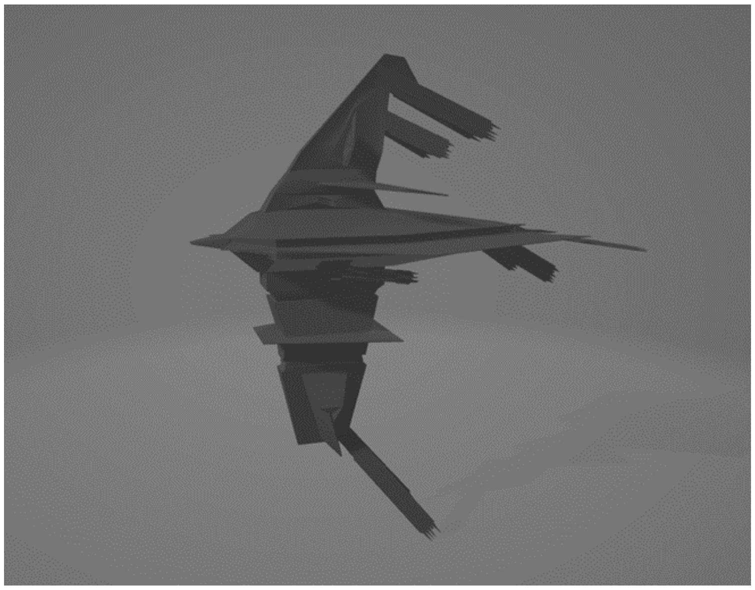


**Figure 51: Divinity Example of Vertical Design**

## 7.22 Buildings

The Imperius faction's architectural layout revolves entirely around a central command center, with each individual structure strategically placed to grant access to vital resources, power, and supplementary units. Conversely, the Divinity faction boasts a main base of operations complemented by multiple ancillary edifices that cater specifically to the player's objectives.

### *7.221 Imperius*

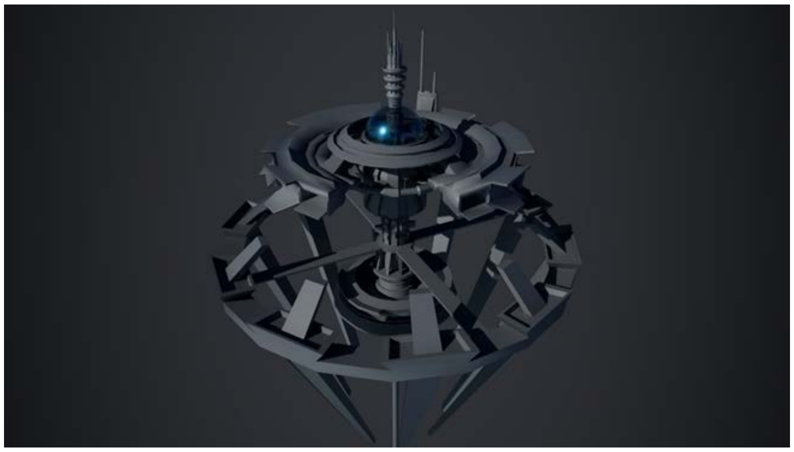


**Figure 52: The Imperius Command Center**

The task of creating architectural designs that are both functional and visually appealing is a complex one that demands extensive investigation. Moreover, constructing the Imperius command center was not only challenging but also ambitious, given that it was to be the largest game model in scale. Consequently, we opted to divide the model into various sections from the apex to the foundation as seen in Figure . This approach enabled us to incorporate more intricacies into the design by segregating distinct features, such as the expansive outer rings and the inner core, which were textured independently.

### *7.222 Animation*

Our objective in developing the game's buildings was to incorporate rudimentary animations that would imbue the inanimate objects with life. For instance, by segmenting the Imperius command center into different sections, we were able to implement a feature where the outer rings of the base rotate during gameplay. We recognized that integrating idle animations into static objects like buildings enhances the dynamic feel of the game, thus making it more immersive.

## 7.23 UV Mapping/Texturing

To texture any 3D model, it is necessary to perform UV mapping, a process that projects the 3D model's surface onto a 2D image, which can then be used for texture mapping. The UV-mapped model and its associated UV image can be imported into texturing software such as Adobe Substance Painter, which is widely used in the industry for creating material images.

The majority of the UV mapping for the Imperius units was automatically done using the "automatic UV mapping" feature in Maya. To improve the image quality, we manually resized certain faces to be more prominent in the UV map. To ensure that different textures could be assigned to the objects, we applied lambert materials to the desired faces. Lambert materials carry basic color data and are one of the primary textures in Maya. By assigning these materials, we could indicate to Substance Painter which parts of the model should be textured separately.

Lambert materials have several advantages in the texturing process. They can allow for faces to overlap each other in the UV map, increasing the size of each face and improving the overall quality of the textures. Additionally, Substance Painter can recognize these materials and use them to distinguish between different texture regions on the model. Overall, the use of automatic UV mapping and manual adjustment, combined with the application of lambert materials, ensured that the texturing process proceeded smoothly and produced high-quality results as seen in Figure .

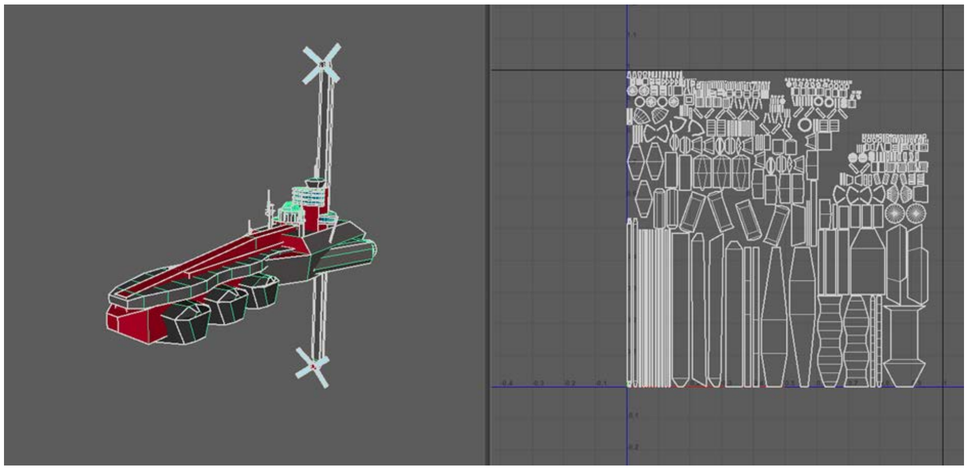


Figure 53: UV Data for the Lodestar Unit

## *7.231 Texturing*

We conducted a series of prototyping and playtesting sessions to ensure model quality and color choices in preparation for texturing the ships. For the Imperius ships, an industrial theme was designed, with a dominant red contrasting with a subtle dark grey as seen in Figure . Meanwhile, the Divinity ships were designed to have a light grey color accompanied by gold lining, with a royal purple hue for the power spikes as seen in Figure . The decision to use high-contrast colors was made to increase the visibility of the ships, especially the smaller ones, in the dark game scenes. The playtesting sessions confirmed that these colors provided the greatest visibility while maintaining a distinction between the Imperius and Divinity ships.

After the UV mapping process, the ships were textured in Substance Painter. The textures were created to follow the selected color schemes for each faction, while also providing an opportunity to add more details to each ship, such as regalia and surface details. In Substance Painter, different texture values were used to affect the overall appearance of each ship. To create an industrial feel for the Imperius ships, a texture with a high metallic value and low roughness value was chosen. On the other hand, the Divinity ships were given reflective textures with high roughness values to further distinguish them from the Imperius ships. finally, each ship was layered with additional textures that added scratches and blemishes to their surfaces for added detail.

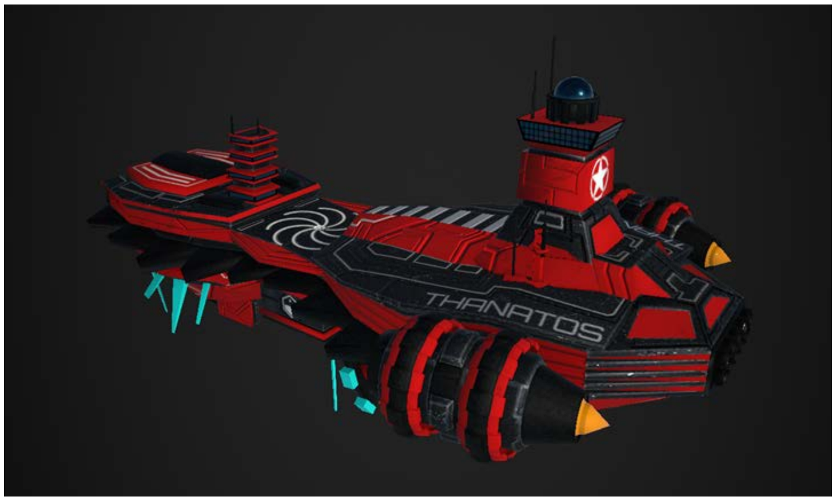


**Figure 54: The Imperius Pacific Dreadnought**

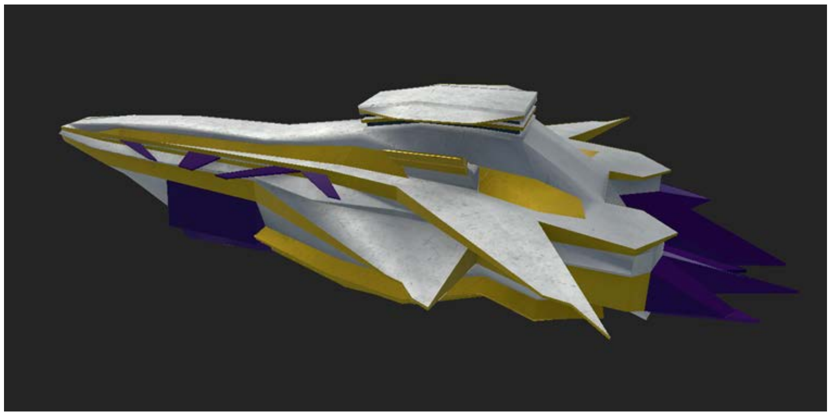


**Figure 55: The Divinity Castillian Cruiser**

## 7.24 Integration

Each model would need to be imported into the Unity engine with its individual UV data and texture images to be implemented and used in-game. Various factors need to be taken into consideration when modeling, UV mapping, and texturing each model to ensure that there would be no errors when integrating them.

### *7.241 Displacement Integration*

When modeling in Maya, each object contains a displacement value relative to the world origin, which is where the X, Y, and, Z values of the rotation and position are zero, and the scale values are one. This means that if an object were to be exported with coordinates not zeroed at the origin, it would carry these values into Unity and displace the object in the game engine. This would then interfere with code that relied on rotations, movements, or scales respective to Unity's world origin. We needed to ensure that each model was properly zeroed at the world origin to have an error-free workflow.

### *7.242 Textures Integration*

When texturing 3D models in Substance Painter, the materials assigned to each face contain a significant amount of data, including information about the height of each face, assigned colors, metallic values, and opacity values. To ensure that all of this data is properly formatted, it is necessary to adjust the export settings in Substance Painter to align with the corresponding Unity presets. Once these textures are exported from Substance Painter, they are saved as PNG files.

Subsequently, the PNG files are imported into Unity and then assigned to their appropriate material slots as shown in Figure . To avoid confusion when assigning materials to the respective models, it is imperative to adhere to a consistent naming convention for each image and material. This practice ensures that each material can be easily identified and assigned to its appropriate 3D model without any confusion or mistakes.

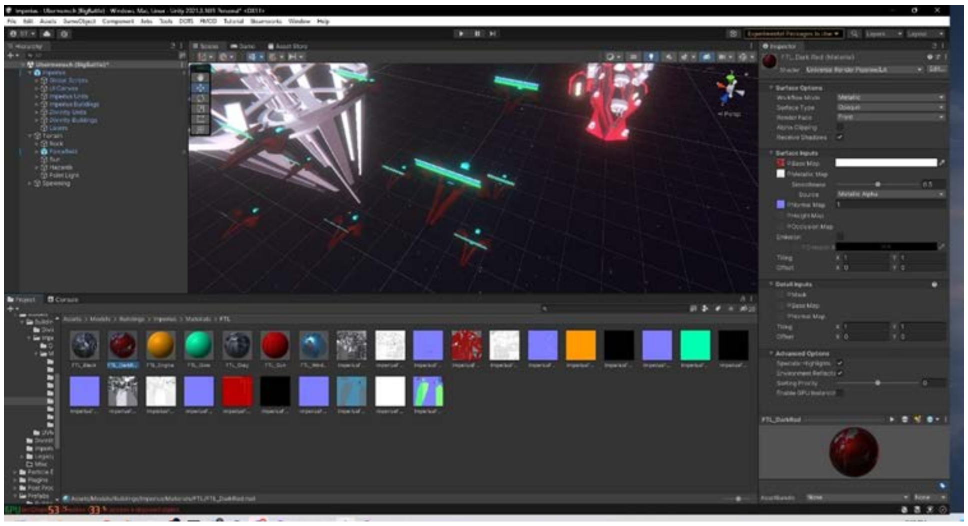


**Figure 56: Texture Images and Their Respective Materials in the Unity Engine**

### *7.243 Prefabs*

Prefabs are a special type of component in Unity that allows for fully configured game objects to be saved in the project for reuse. By using prefabs, we were able to make changes to models more efficiently when assigning new texture sets, applying new models, or editing a ship's properties.

## 7.26 Modelling Challenges

Throughout the development of Imperius, the team faced and overcame many challenges in the art design cycle. These challenges taught us many lessons about creating models to professional standards and helped us understand the importance of a solid content creation pipeline.

### *7.261 Single Object vs. Multi-Object Modeling*

In the course of ship modeling, a dilemma emerged whether to create ships using a single object, such as a cube with multiple extrusions or to use a combination of multiple objects. This decision has significant implications not only for the visual quality of the ships, but also for the UV mapping of the objects, and the eventual integration of the models into the Unity game engine. Specifically, using a single object may simplify the modeling process, but may result in a less detailed final product. Conversely, utilizing multiple objects may allow for more intricate and realistic designs, but may increase the complexity of the modeling process as shown in Figure .

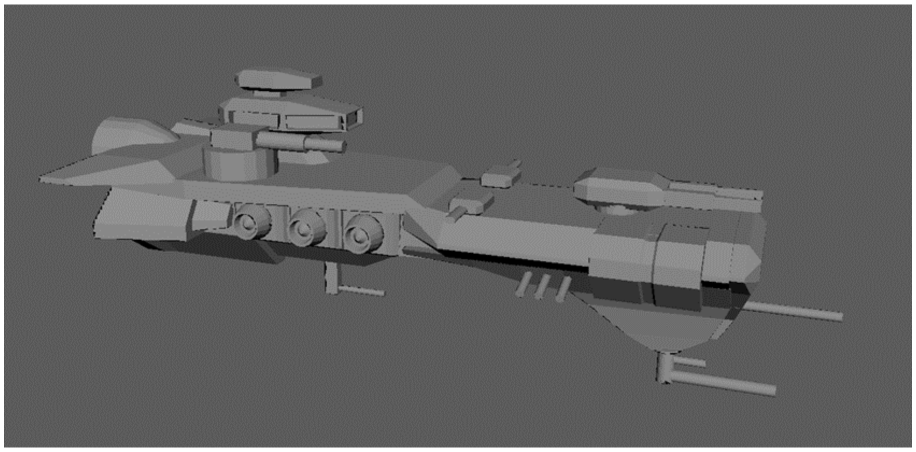


Figure 57: The Verdancy Battle Cruiser Showing Multi-object Modelling

Most of the larger Imperius ships modeled were created using multiple objects, which allowed for more interesting and unique shapes. As we continued to model out larger and larger ships, we realized that it was in our best interest to use less extrusions and more objects. This is because the detailing on larger ships was small enough that only using extrusions from a single object would not provide enough detail, as shown in Figure 58 and Figure 59.

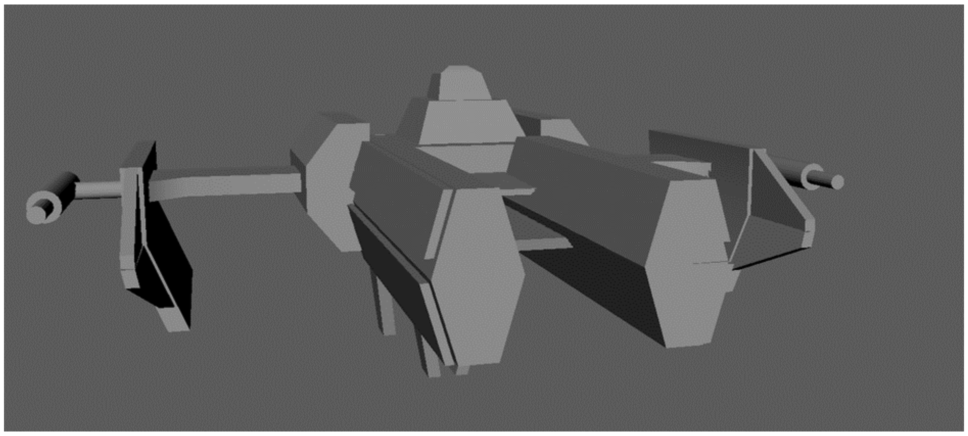


Figure 58: The Imperius Carrier Ship

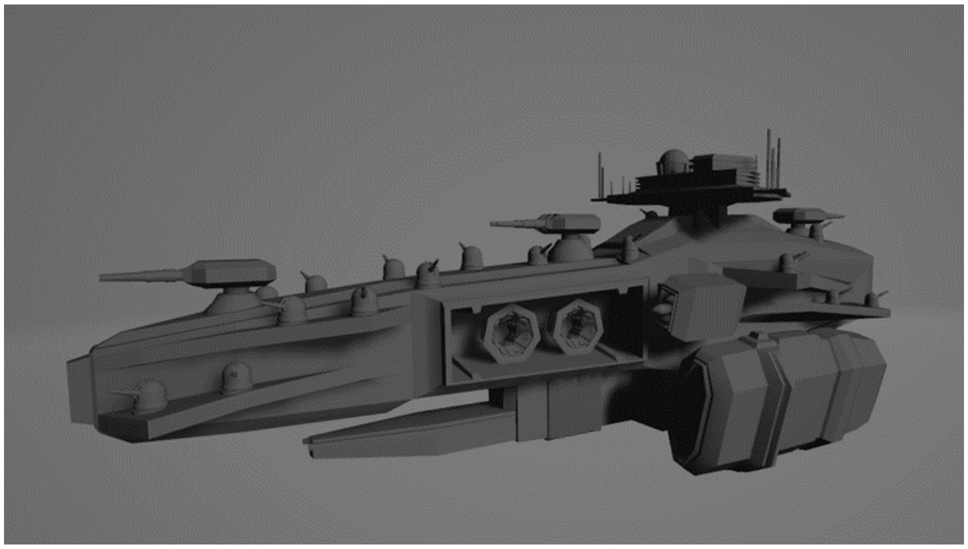


Figure 59: The Imperius Battleship Aurelius

### 7.262 Symmetry Issues

Due to the decision to move away from the overuse of symmetry while modeling ships, some had issues with object faces not being recognized by Maya, which would result in face values being lost. Furthermore, switching between local and global coordinate systems led to mismatching displacement data for these faces, which would lead to implementation issues in the game engine. To solve these problems, we either used the "Freeze Transformations" tool in Maya to reset each object's transformation, or we manually changed the transformations in Unity.

### 7.263 UV Issues

There were multiple problems regarding the relationship between the size of ships and the size of faces in UV maps. As the size of a ship increased, the size of each individual face in the UV map decreased in size as seen in Figure 60. This meant that larger ships would have decreased quality when transferred into Substance Painter.

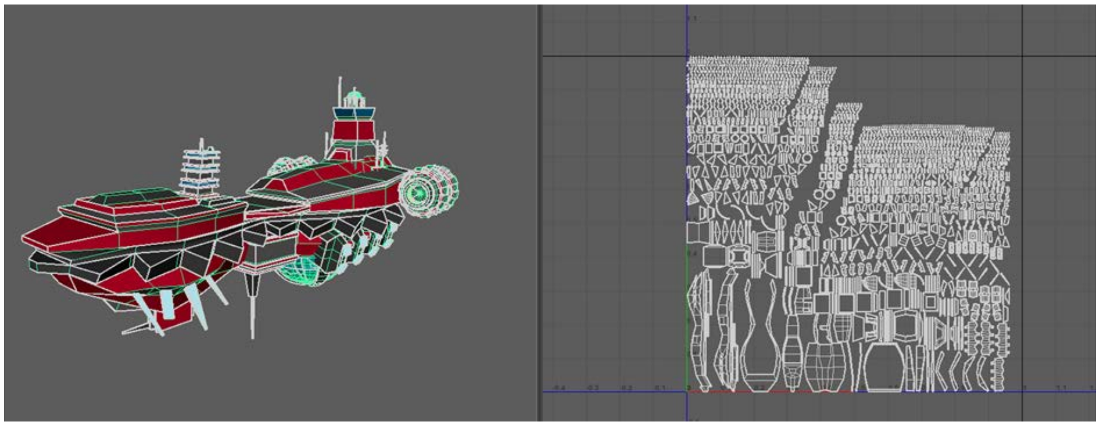


Figure 60: UV Data for the Pacific, the Imperius Super Capital Ship

This is where the ‘single versus multiple objects’ problem from modeling came into play. If a ship was assigned multiple objects, each object would have to be UV mapped separately before combining them into one object. This tedious process resulted in many ships containing incorrect UV mappings. On the other hand, single-object ships would contain extremely small faces in their UV maps, leading to a loss in detail.

#### *7.264 Texturing Challenges*

Substance painter treats all objects as the same size, regardless of their individual dimensions when importing them into the same project. This affected the resolution of textures since larger ships and smaller ships were given the same texture limit, which meant that larger ships were given less room to add detail. Additionally, due to the limitations of some textures, trying to increase the tiling values made them look worse than if they were left in their original state. On the other hand, textures detailed in Substance Painter would not be guaranteed to look the same in engine with different lighting and rendering settings.

The challenges encountered in implementing modifications to the texture of a ship were quite time-consuming, as they required modifications across multiple software platforms including Substance Painter, Maya, and Unity. For instance, if a new texture was required for a ship, it would demand the creation of new UV maps, which would involve modifications to either the UV map or the materials allocated to the faces. Subsequently, the newly created UV maps would have to be imported back into Unity, and thoroughly tested in the engine and game to ensure a seamless integration. This workflow required a significant investment of time and effort to guarantee that the final product was polished and of high quality.

## 7.3 VFX

In Imperius, the VFX design played a multifaceted role, contributing significantly to the game's overall visual appeal and gameplay experience. One of the notable ways we utilized VFX was by leveraging Unity’s particle system collision module, which allowed us to add de-facto multithreading to our weapon systems. Most of the VFX assets in the game were either heavily modified from online sources or designed from the ground up to suit the game's style and theme. (Figure 61 - Figure 66)

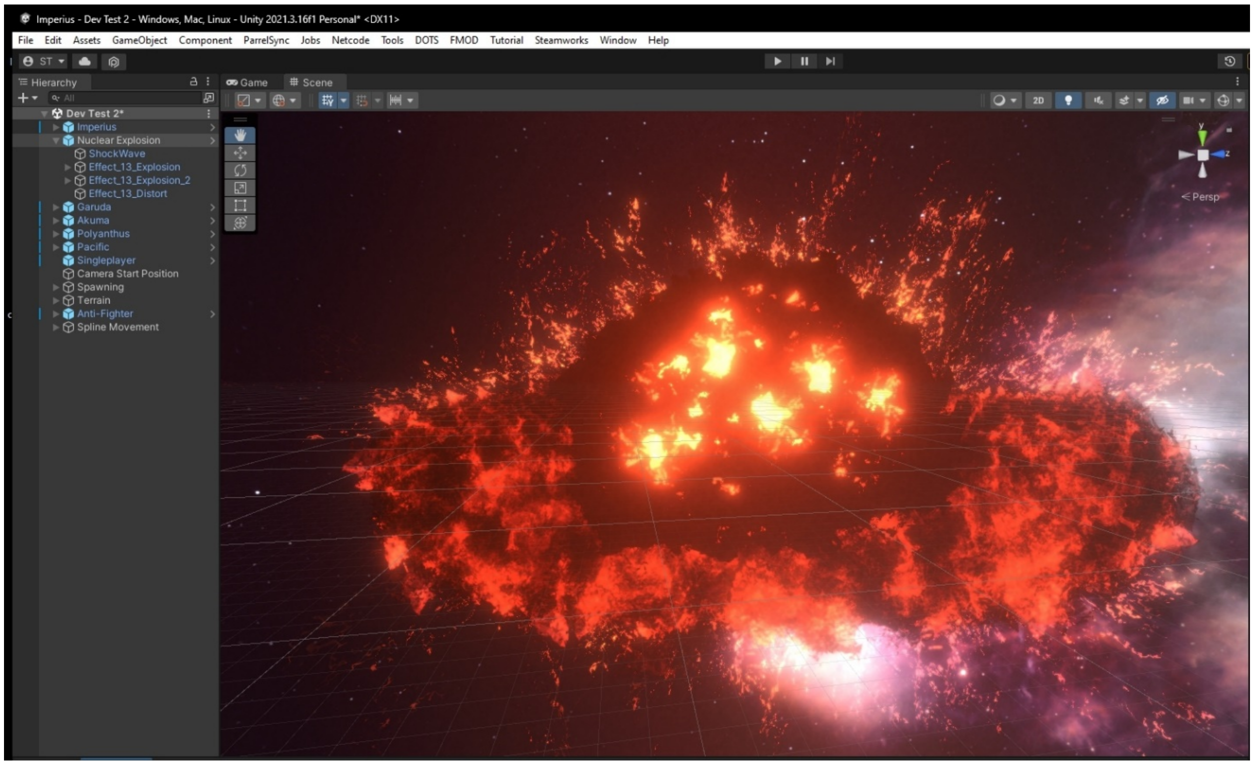
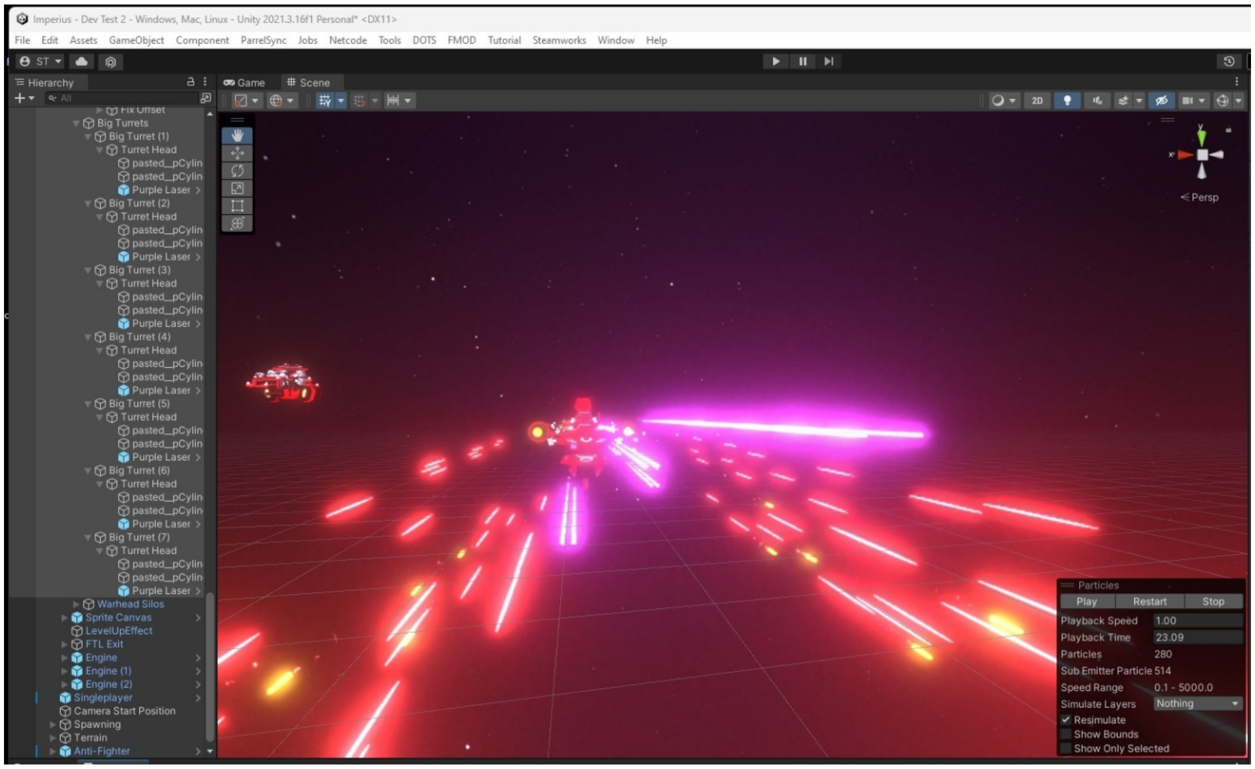


**Figure 61: A Nuclear Explosion VFX**



Playback Speed 1.00
Playback Time 23.09
Particles 280
Sub Emitter Particle 514
Speed Range 0.1 - 5000.0
Simulate Layers Nothing
✓ Resimulate
Show Bounds
Show Only Selected

**Figure 62: Laser Barrage from a Pacific**

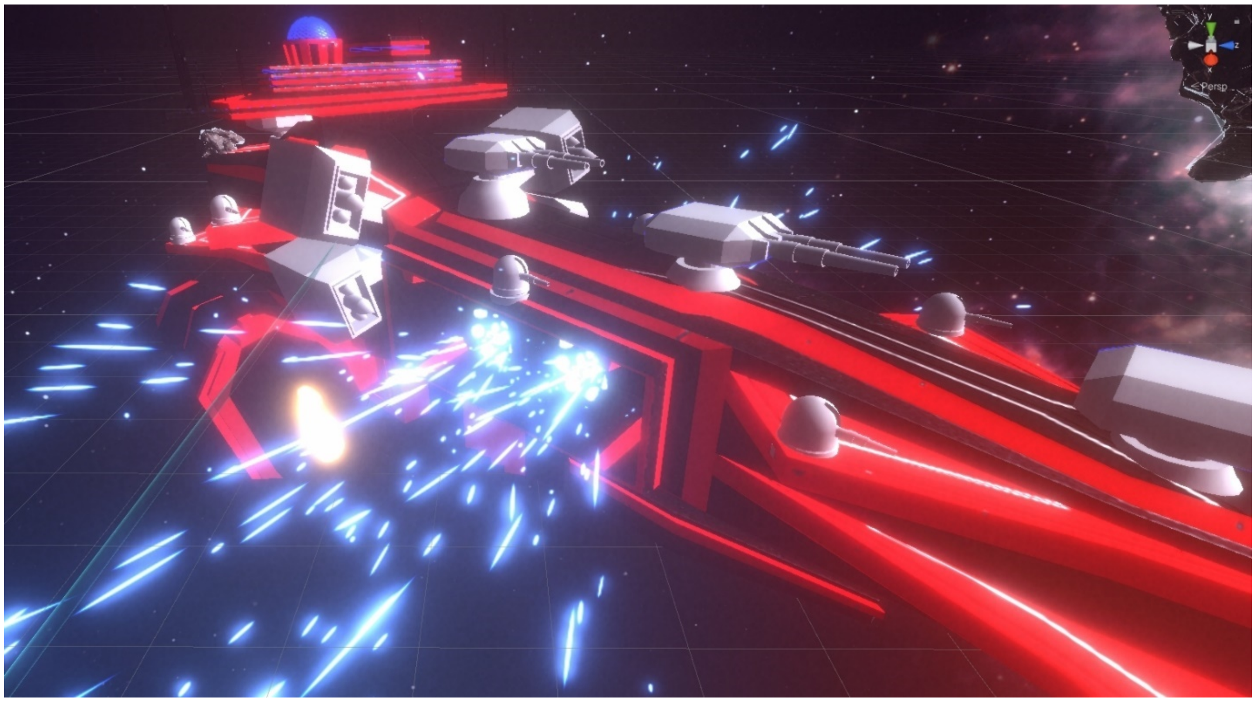


Figure 63: Aurelius Charge Up Ability

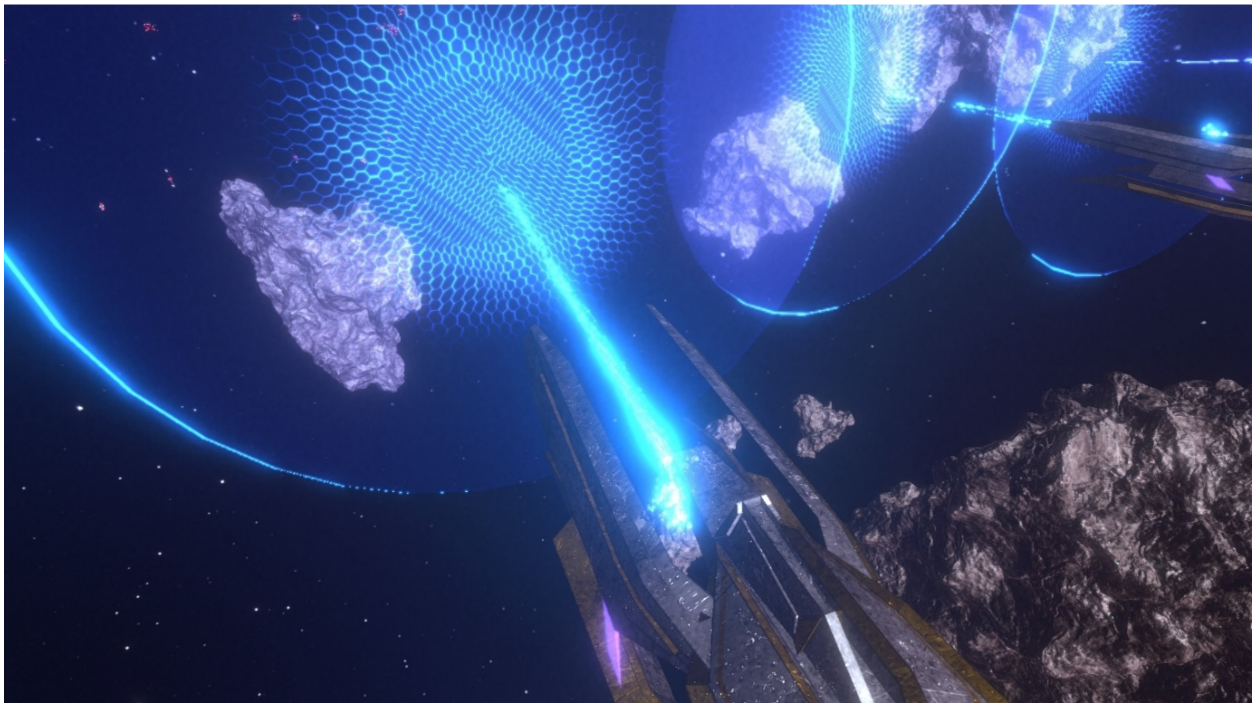


Figure 64: Divinity Force Field

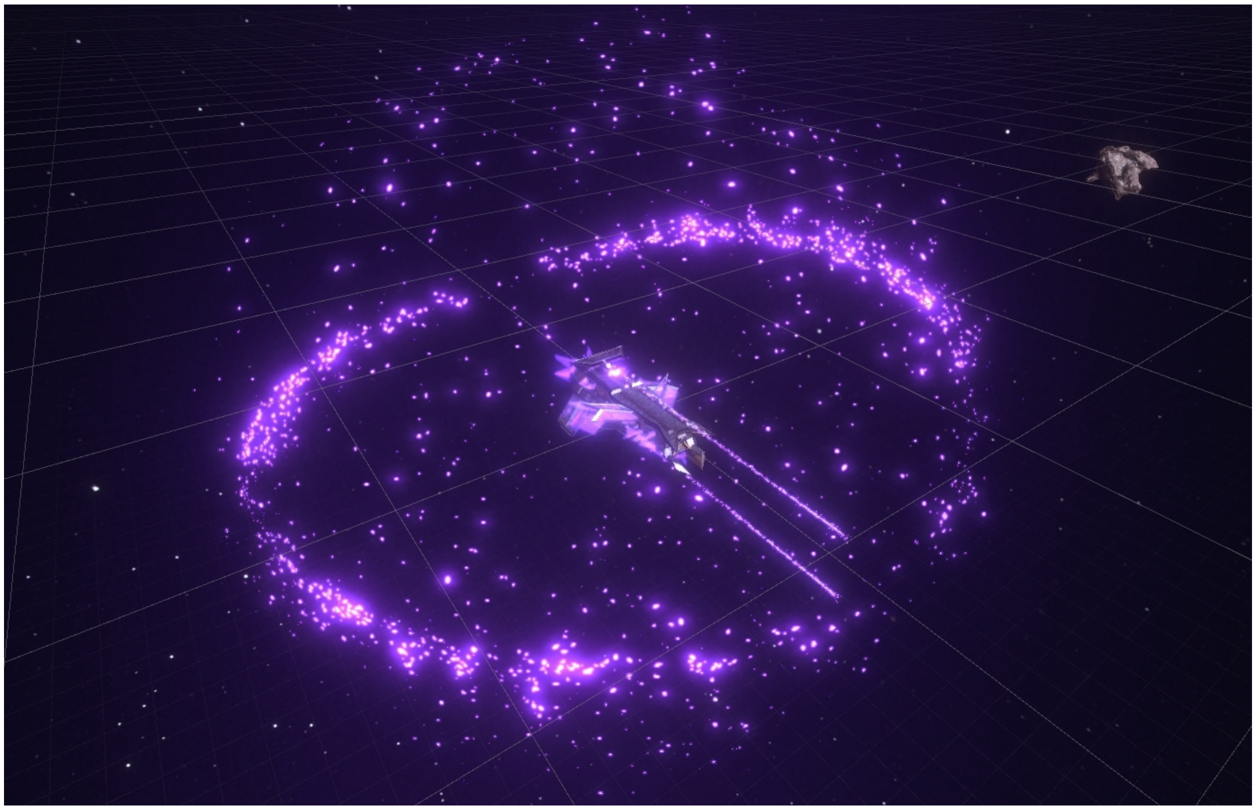


Figure 65: Divinity FTL Inhibitor Effect

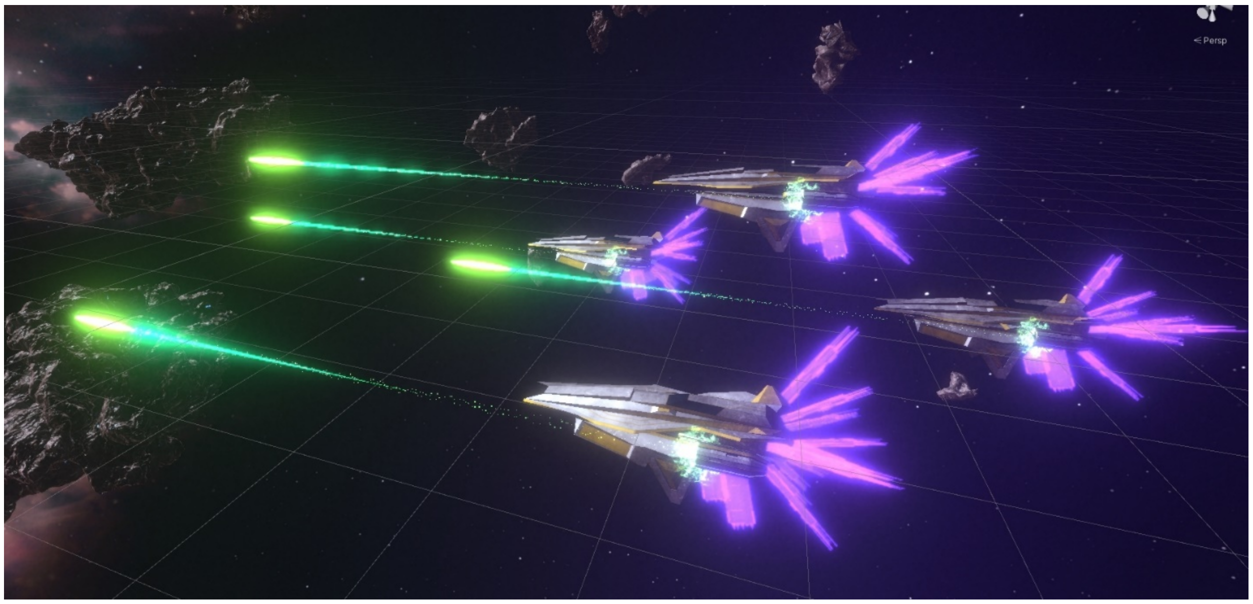


Figure 66: Divinity Artillery Units

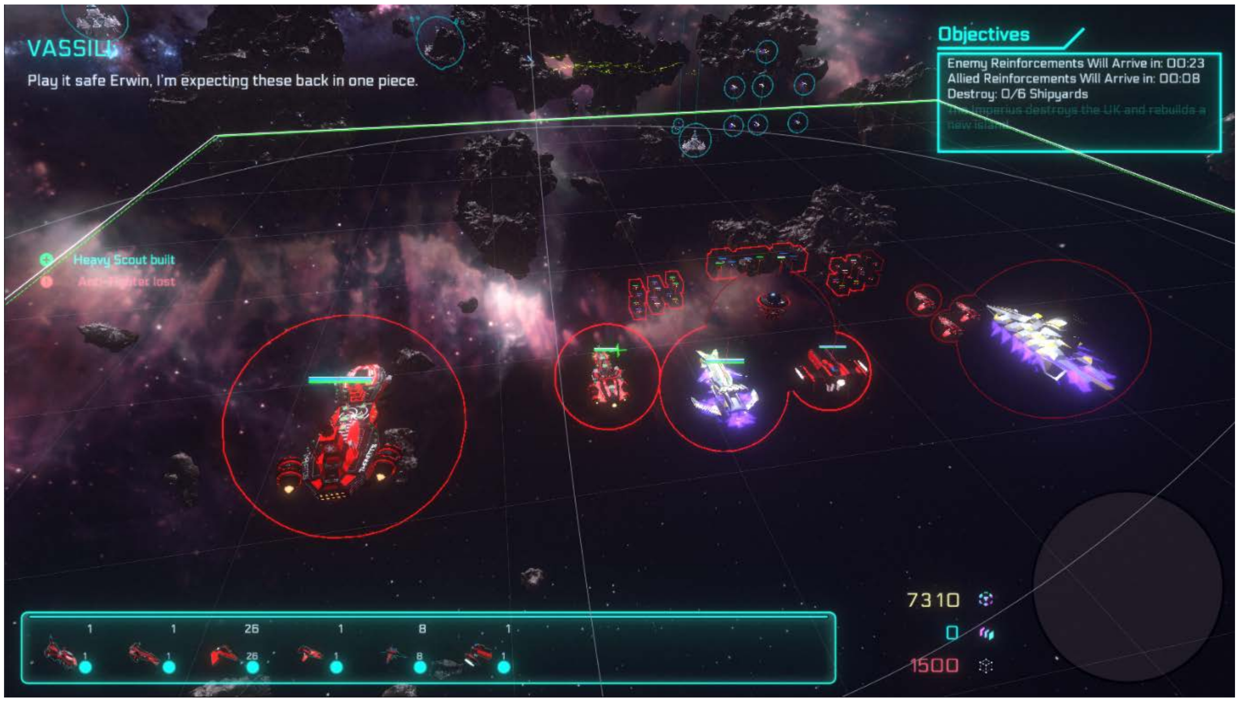
We took a strategic approach to designing VFX for the different factions in the game. The Divinity and the Imperius each have unique VFX styles that complement their respective roles in the game. The Imperius, being humans, rely on more "modern" weapons, such as bullets and missiles, resulting in VFX with a more familiar, realistic feel. On the other hand, the Divinity use more futuristic effects and outlandish colors to convey their otherworldliness, creating a sense of awe and wonder in the players.

## 7.4 User Interface

Our objective in designing Imperius' user interface (UI) was to create a simple, yet elegant interface that would not hinder gameplay for the player. To achieve this, we utilized the Nova UI asset package, which offers a comprehensive set of tools that build upon the existing Unity UI system.

During gameplay, certain critical information needs to be displayed to the player to ensure success. In Imperius, the HUD played a vital role in displaying this information, which included current objectives, the number and type of ships selected, resources, power, and tech level. Furthermore, the HUD was also responsible for displaying dialogue and popup messages to alert the player of critical information about the current game state.

To create an effective HUD, we started by gathering research data and sketching out various ideas to determine the best way to present this information. Once we arrived at a finalized design, we leveraged our knowledge of Nova UI to implement our concept into a functional system. By adopting this approach, we were able to create a HUD that was both informative and aesthetically pleasing, while avoiding any interference with gameplay as seen in Figure 67.



Play it safe Erwin, I'm expecting these back in one piece.

Enemy Reinforcements Will Arrive in: 00:23
Allied Reinforcements Will Arrive in: 00:08
Destroy: 0/6 Shipyards
~~The Imperius destroys the UK and rebuilds a new island~~

Heavy Scout built
Anti-Fighter lost

Figure 67 : A Screen Capture Displaying the HUD

Once the HUD was complete, our next task was to design the pause and building menus. For the pause menu, we opted for a straightforward, commonly used design where all the necessary buttons were situated at the center of the screen for easy access. The pause menu can be seen in Figure 68.

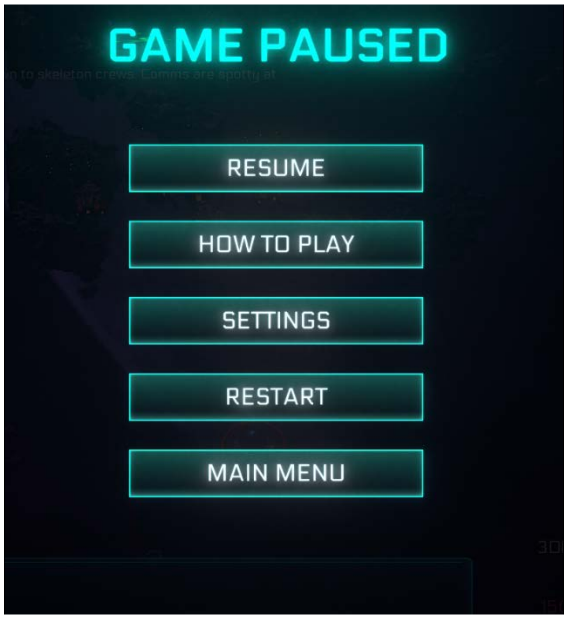


**Figure 68: The Pause Menu**

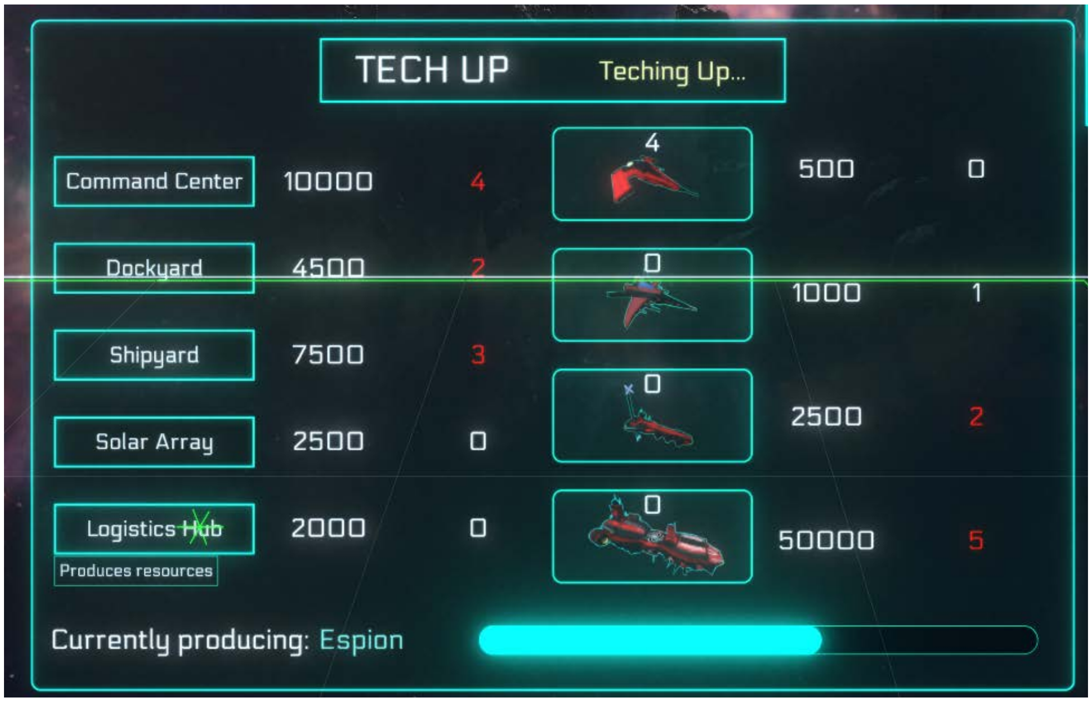


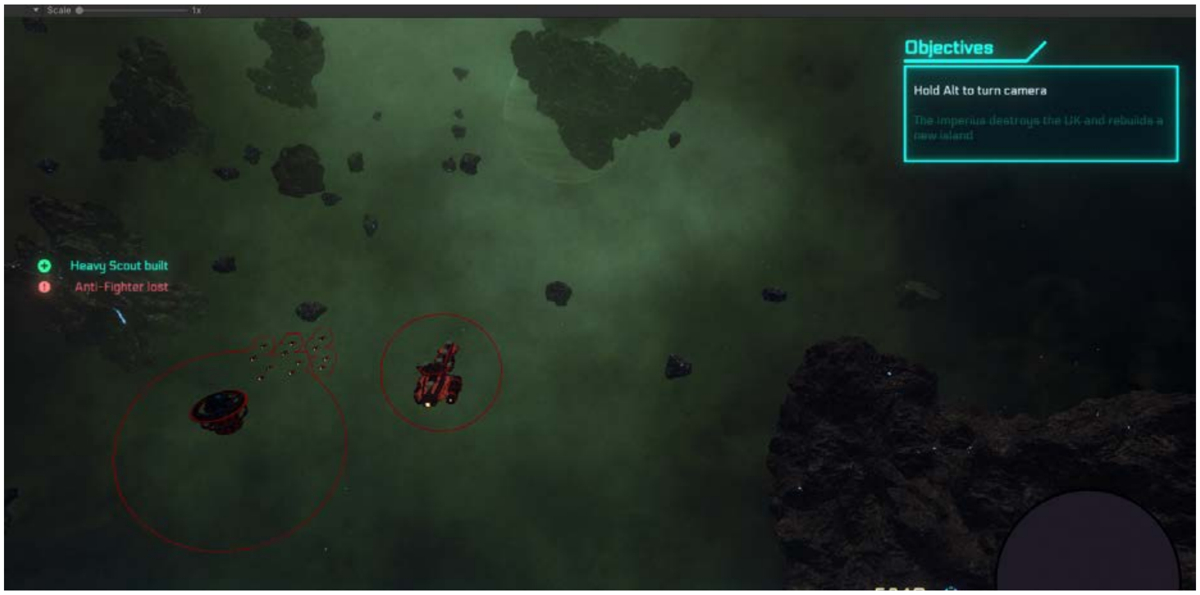
**Figure 69: The Building Menu for the Imperius Main Base**

The building menu, shown in Figure 69, was a bit more complex as it needed to provide the player with information related to ship creation, resource management, and building creation. As such, we designed the menu to be highly informative, providing players with all the necessary information at a glance. By doing so, we ensured that players could make informed decisions about their next moves, without the need for extensive menu navigation. Overall, the creation of these menus complemented our design philosophy of keeping the interface simple, yet effective.

## 7.5 Lighting/Post Processing

In space, proper lighting and post-processing are crucial to create an immersive environment. It was important to ensure that the ships and visual effects stood out against the dark background, while still maintaining a sense of realism. To achieve this, we utilized Unity's Universal Render Pipeline (URP), which provided us with the necessary tools to fine-tune the lighting and post-processing.

To create a unique setting and atmosphere for each level, we used multiple directional lights to light up the scene. These lights were placed outside the game world and were used as an ambient light to represent light sources such as distant stars and galaxies. In addition, we made use of post-processing tools provided by URP to enhance the visual effects and improve the visibility of UI elements.



Hold Alt to turn camera

Heavy Scout built

Anti-Fighter lost

Figure 70: The Pax Imperia Level

Each level in Imperius has a unique background, ranging from galaxies to dark, misty asteroid fields such as the one shown in Figure 70. Each level contained a unique setting and atmosphere, which required varying lighting settings. Apart from the directional lights used in the scene, we also made use of post-processing tools provided by Unity’s URP. We used these post-processing tools to help the ships pop out in the background and to make some UI elements easier to see.

To further enhance the lighting in Imperius, we utilized baked lightmaps, which pre-calculate the brightness of surfaces in a Unity scene. However, due to the nature of the game, where nearly every object on the screen is in motion, we could only use baked lightmaps for promotional screen captures of the game as seen in Figure 71. Nonetheless, we were able to utilize various lighting techniques to create a visually stunning game.

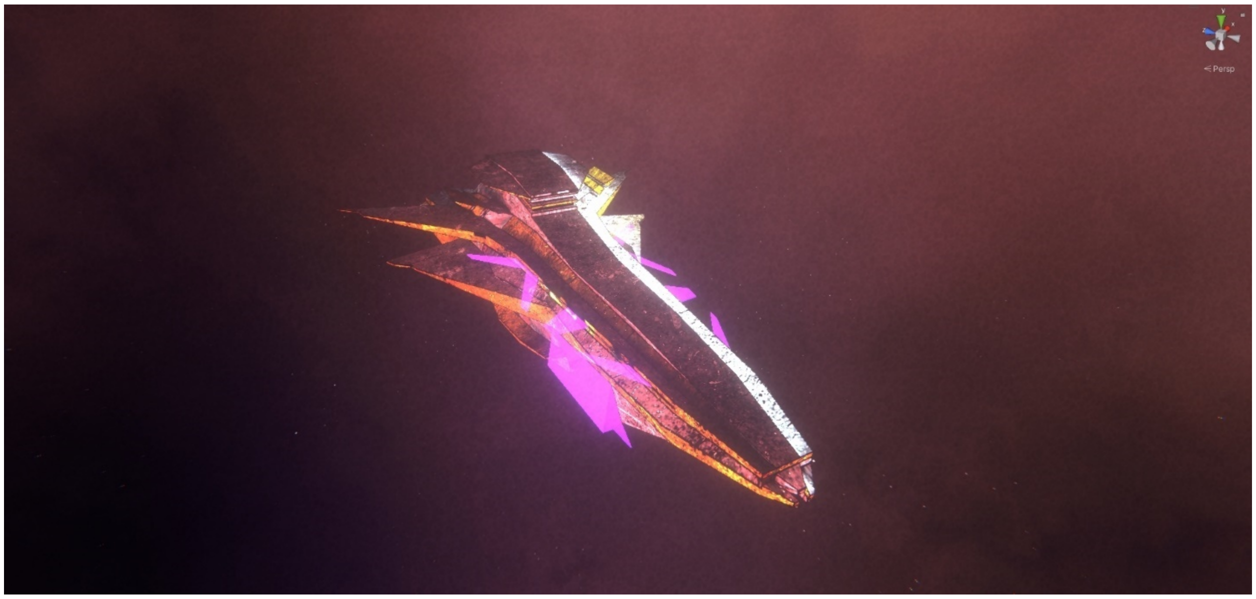


Figure 71: Baked Lightmaps on a Divinity Ship

## 7.6 Cinematics

For the creation of the game trailer's cinematic, the team utilized both Cinemachine and the Unity recorder to capture video of the cameras set up within the game engine. A significant amount of time was devoted to finding a way to record cinematic footage from a separate camera while also capturing footage from the gameplay perspective. This was done to ensure that the trailer could showcase both the game view and the cinematic shots to provide a complete picture of what was happening as seen in Figure 72 and Figure 73.

Figure 72: A Cinematic Shot from the PAX East Trailer

Figure 73: A Gameplay Shot from the PAX East Trailer

An initial skeleton of the video was then produced in Adobe Premiere, with only text describing the shots that were planned. This step enabled the team to get a better understanding of how the final product would look and allowed synchronization of the transitions with the music. After the skeleton was complete and gameplay footage had been recorded, the final product was assembled in Adobe Premiere. The trailer featured a combination of in-game footage, voice acting, and background music to highlight the various gameplay elements such as combat, base building, and unit production.

# 8. Audio Design

## 8.1 FMOD Audio Design

We utilized the audio middleware program FMOD to implement all the SFX and music in the engine, as it has many advanced features that allow for a dynamic audio experience during gameplay. One of the major features used was the ability to randomize the pitch of sounds. We pitched all ship, weapon, and ability sound effects randomly up and down within a set range, making them less monotonous when heard repeatedly. We also implemented a linear distance-volume curve, which caused every sound effect to soften as the player moves the camera further away from the source of the sound.

Another major FMOD feature we utilized was custom variables. We created an “intensity” variable, defined as any floating-point value ranging from 0 to 10 inclusive depending on the number of lasers currently firing and the number of ships that have been destroyed in a predefined interval. With this variable, we were able to implement three groups of songs and three groups of background voice lines, which we labeled Low, Medium, and High intensity to play at different intensity values. During gameplay, FMOD plays voice lines at a steady rate, which increases in frequency as the intensity of the battle increases. The order of voice lines is chosen at random from the appropriate intensity group. FMOD also chooses one of two appropriate-intensity songs to play over a predefined time interval, with occasional breaks in the music to make the soundscape less repetitive. Whenever the intensity changes, the music must transition from one song to another, so we implemented transition timelines that would slowly fade out the current song and then slowly fade into the next song, to make the transition as smooth and subtle as possible. The implementation of this intensity system is shown in Figure  below. The game also plays ambient noises at random intervals intended to imitate the inside of a spaceship and the feeling of space.

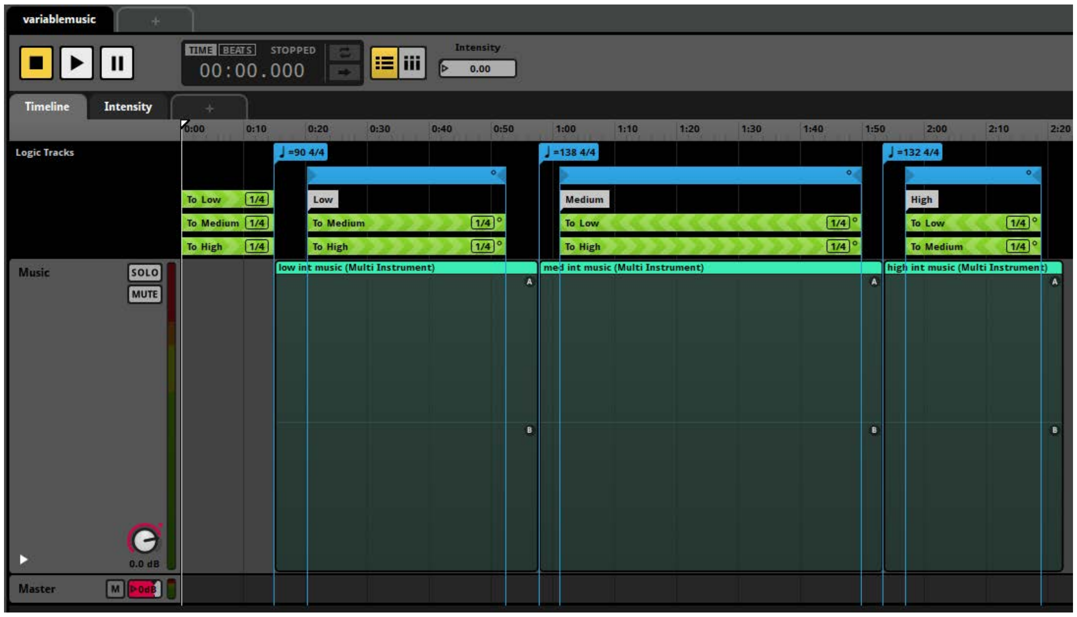


Figure 74: A screenshot of the variable music event in FMOD.

We also had to separate the sounds into “buses” or audio channels that each contain their own category of sound. We created 6 buses: music, sound effects, ambient noises, background dialogue, character dialogue, and story dialogue. This lets us apply audio effects to an entire category of sound at once and also allows the player to increase or decrease the volume of each category separately. For the background dialogue, we used a frequency equalization (EQ) effect to reduce the low and high frequencies of the voices, to make it sound as if the dialogue is spoken through a radio. For the ambient noises, we applied a large amount of reverb to make the noises extremely subtle.

## 8.2 FMOD Integration

Integrating FMOD events into Unity was no simple task and required numerous custom systems to function properly. Given the fact that dozens of lasers can fire per frame, memory constraints as well as audio peaks need to be considered. We designed a pooling system integrated with a priority queue where weapons were prioritized based on distance, size, and other factors.

```
function RecalculatePriorities():

      while not crossedIteration:
          if forwards:
              currWeapon = audioWeapons[currForwardIndex]
              if currWeapon.isFiring and currWeapon.needsEmitter:
                  requiresEmitter = currWeapon
                  forwards = false

              currForwardIndex = currForwardIndex + 1
          else:
              currWeapon = audioWeapons[currBackwardIndex]
              if currWeapon.isFiring and not currWeapon.needsEmitter:
                  currWeapon.ReleaseEmitter()
                  requiresEmitter.TryTakeEmitter()
                  forwards = true

              currBackwardIndex = currBackwardIndex - 1

          if currForwardIndex == currBackwardIndex:
              crossedIteration = true
```

**Figure 75: Pseudocode for Redistributing Audio Emitters**

This allowed us to experiment with a certain number of sound emitters, and find the right balance of weapons fire to complement the soundtrack, voice lines, and other sounds. While not perfect, this solution (Figure ) is certainly efficient, utilizing a mere O(1) space complexity and O(n) time complexity.

## 8.3 Music Design

All songs were composed in the music program Ableton, using MIDI instruments. We picked out several free packs of instruments from various music websites and recorded melodies and rhythms using a MIDI keyboard. One especially helpful instrument pack was the BBC Symphony Orchestra from Spitfire Audio's website, as it contained almost every instrument needed to make orchestral battle songs. After creating melodies, chord progressions, basslines, and drum loops for each song, we moved onto the mixing stage, which consists of adjusting the volume levels of each instrument so that no instrument is too overpowering or too quiet. We also applied various audio effects to each track, including reverb, delay, EQ, distortion, and compression.

Due to the "intensity" variable discussed in the section above, players would not be listening to the same song for longer than a short, predetermined interval, so most of the songs were composed with this short timeframe in mind. Songs also had to loop smoothly, so that players would not be able to tell that a song had suddenly started over during gameplay. To follow these constraints, we used short, memorable melodies that were repeated often to convey the desired emotion quickly. Finally, songs had to fit within their given intensity group, and so overall volume and the number of instruments in each song were adjusted such that no song would seem out of place.

## 8.4 Sound Effect Design

We lacked the resources to do proper Foley sound effects, so most of our sound effects were originally taken from the royalty-free sound effect libraries Freesound.org, pixabay.com, and mixkit.co. Sounds that were not taken from libraries were either synths that were created in Ableton or brief recordings from our personal microphones. Using the sound editing programs Audacity and Ableton, we applied various audio effects to each sound to achieve a cohesive, futuristic, and dramatic soundscape for the game. The sounds for each type of laser that could be fired had to be differentiated from each other, so we pitch-shifted each laser either up or down until they were easily distinguishable for the players. Almost all of the ability sound effects from the sound libraries were too quiet, so we had to apply compression to make the sounds perceivably louder without making them peak in volume. The explosion sound effects were the opposite: far too ear-piercing and noticeable, so we applied reverb to make them softer and more subtle. Certain sounds also had to be sped up or slowed down to match their in-game implementation.

## 8.5 Voice Lines

To get a cast of voice actors, we reached out to friends and recruited volunteers from WPI's theater department to audition. Some voice actors recorded voice lines on their microphones and then uploaded them to Onedrive, but the majority of actors met with us in person to record their voice lines.

We made several preparations to ensure that the in-person recordings were as clean and high-quality as possible. First, we left all our sound-producing objects outside of the recording studio. Secondly, we adjusted the microphone to account for the voice actors' varying heights. Thirdly, we asked the voice actors to sit up straight, hold the script behind the microphone, and read their voice lines through the pop filter so that their heads are pointed straight forwards, to allow for the maximum amount of breathing room in their chest and stomach. Some voice actors preferred to stand rather than sit while performing their lines, as shown in Figure 76.

We conducted two rounds of in-person recording sessions. The first round had two purposes: to serve as auditions so we could hear which actors had the best performances and secondly to record some character barks that would play randomly in the background during gameplay. After the first round was over, we listened to the recordings and picked out who would play each character. Our criteria were based on which actors could convey emotions best, and whose voices fit the characters best.

Figure 76: A Voice Actor Reading Lines

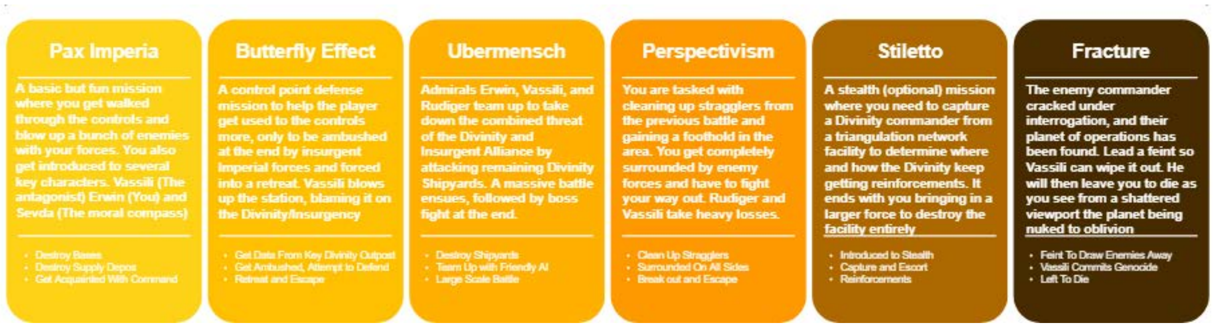
Following the completion of the casting process for all characters, the project moved forward to the second round of voice acting. The objective of this stage was to record the script for the main story, which included numerous conversations between various characters. However, due to conflicting schedules among the participating students, it proved to be a daunting task to schedule multiple voice actors to be present in the same room simultaneously. As a result, single-person recording sessions were conducted instead.

To ensure that the voice actors grasped the intended mood and energy of the conversations they were recording, an additional team member was present during the recording sessions alongside the audio engineer. This individual read the other half of the conversation to the actor as they performed their lines, providing necessary cues and serving as a sounding board for the actor. This approach facilitated the recording process and resulted in more consistent performances from the voice actors.

Occasionally, voice actors faced challenges when it came to performing particularly verbose lines in the script. As a solution, we allowed them to modify the wording slightly while ensuring that the changes did not alter the intended meaning of the dialogue. This approach helped to ensure the quality and coherence of the recordings.

# 9. Level Design

We decided to create a total of 6 levels for the campaign, as can be seen in the table below. When designing these levels we took a holistic approach, taking into consideration the campaign as a whole and ensuring that each level was a unique experience from the other that each highlighted various elements of the gameplay. This was achieved through group brainstorming sessions where we also hashed out the narrative. This was to establish a solid foundation and what key experiences we wanted out of each level. Soon after began the individual iterative prototyping process. This process entailed the following: designing a level prototype around the key experience we wanted to achieve, playtesting it for bugs, then finally integrating it with the narrative such as adding dialogue and mission details.



| Pax Imperia | Butterfly Effect | Ubermensch | Perspectivism | Stiletto | Fracture |
|---|---|---|---|---|---|
| A basic but fun mission where you get walked through the controls and blow up a bunch of enemies with your forces. You also get introduced to several key characters. Vassili (The antagonist) Erwin (You) and Sevda (The moral compass) | A control point defense mission to help the player get used to the controls more, only to be ambushed at the end by insurgent Imperial forces and forced into a retreat. Vassili blows up the station, blaming it on the Divinity/Insurgency | Admirals Erwin, Vassili, and Rudiger team up to take down the combined threat of the Divinity and Insurgent Alliance by attacking remaining Divinity Shipyards. A massive battle ensues, followed by boss fight at the end. | You are tasked with cleaning up stragglers from the previous battle and gaining a foothold in the area. You get completely surrounded by enemy forces and have to fight your way out. Rudiger and Vassili take heavy losses. | A stealth (optional) mission where you need to capture a Divinity commander from a triangulation network facility to determine where and how the Divinity keep getting reinforcements. It ends with you bringing in a larger force to destroy the facility entirely | The enemy commander cracked under interrogation, and their planet of operations has been found. Lead a feint so Vassili can wipe it out. He will then leave you to die as you see from a shattered viewport the planet being nuked to oblivion |
| • Destroy Bases • Destroy Supply Depot • Get Acquainted With Command | • Get Data From Key Divinity Outpost • Get Ambushed, Attempt to Defend • Retreat and Escape | • Destroy Shipyards • Team Up with Friendly AI • Large Scale Battle | • Clean Up Stragglers • Surrounded On All Sides • Break out and Escape | • Introduced to Stealth • Capture and Escort • Reinforcements | • Feint To Draw Enemies Away • Vassili Commits Genocide • Left To Die |

**Figure 77: Simplified Level Overview**

There were some struggles initially, with limited assets and the jump from 2D to 3D made the design of the levels a challenge. That in addition to the limited top-down camera view made it difficult to effectively showcase a backdrop or scenes occurring in the background without having everything be below the camera which we wanted to avoid doing for every level to not be repetitive. Despite this, by focusing on the desired experience objective first, we were able to accomplish our level design goals by designing around said objective.

## 9.1 Pax Imperia

Pax Imperia acts as a tutorial for the game that walks the player through the controls and ends with destroying an enemy ship. Our goal for Pax Imperia as the first level was to draw the player into both the gameplay and narrative. To do this it was decided to use striking and moody visuals to convey the tone of the upcoming story and get the player's attention. However, this level was in a limbo state before the narrative was finalized, its objective going from destroying several bases to destroying a single grouping of ships. To best fit with the narrative we played around with sight lines, and as a part of the tutorial, the player is instructed to move to a certain area where there is a gap in the asteroids and allows an unobstructed view of the planet in the background in time with the narrative revelation that this is the Divinity's home planet.

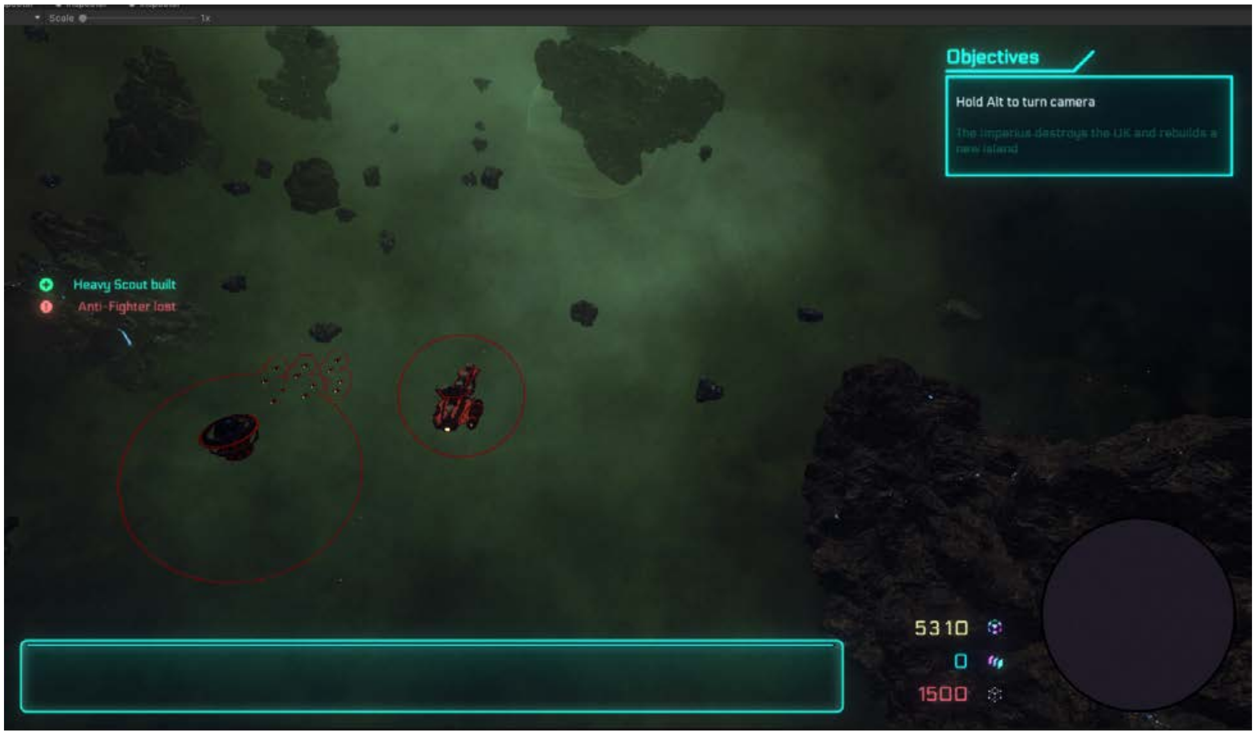


**Figure 78: First View in the Game**

## 9.2 Butterfly Effect

Butterfly Effect is a mission where the player must defeat opposing ships and then control a point for an extended period. Our goal with Butterfly Effect was to engage the player with the idea of capturing an area and defending two points at the same time. These two points are the communication tower and your main base, first the player must use their skills learned in the first mission to drive out enemy combatants, then they must learn how to create an efficient production method to continuously pump out ships against enemy reinforcements that are trying to retake the point they captured as well as attack their base. Sightlines were also an idea toyed with in this level as the use of large asteroids on the side and smaller ones in the middle creates a lane of sorts from the base to the capture point, a lane that the player needs to defend to ensure their reinforcements can reach their front-line units.

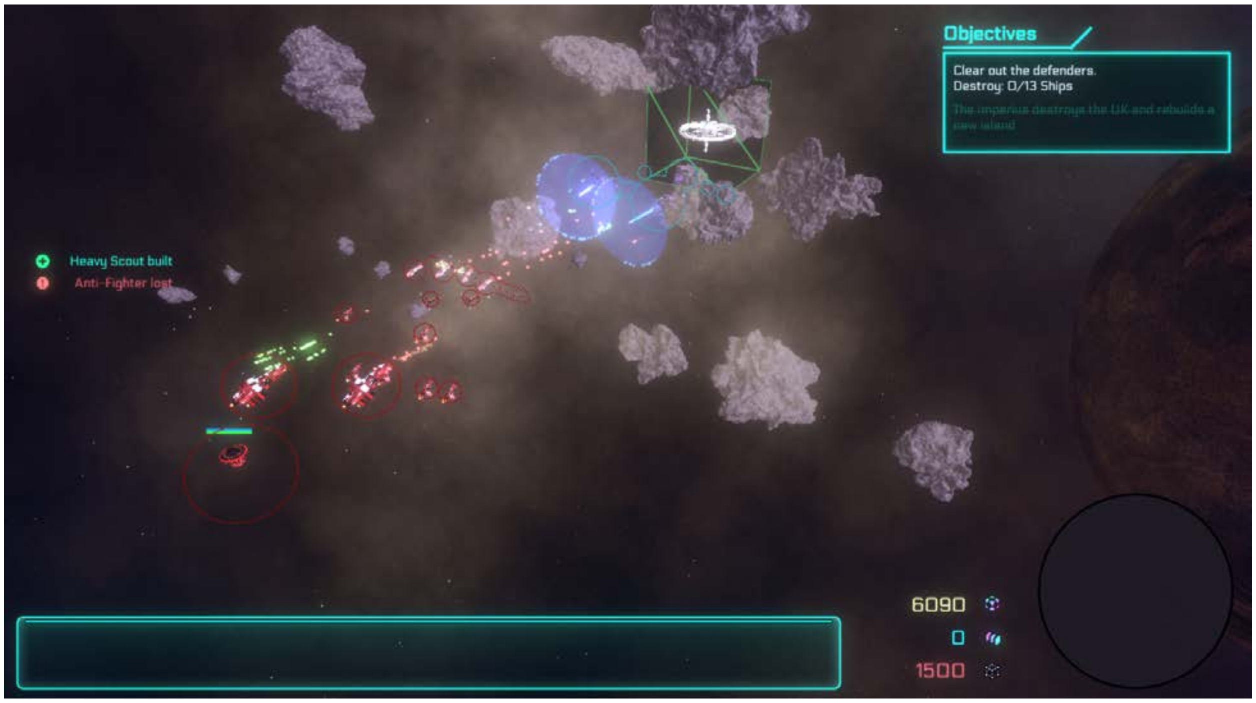


Figure 79: Lane the Player Must Clear and Defend

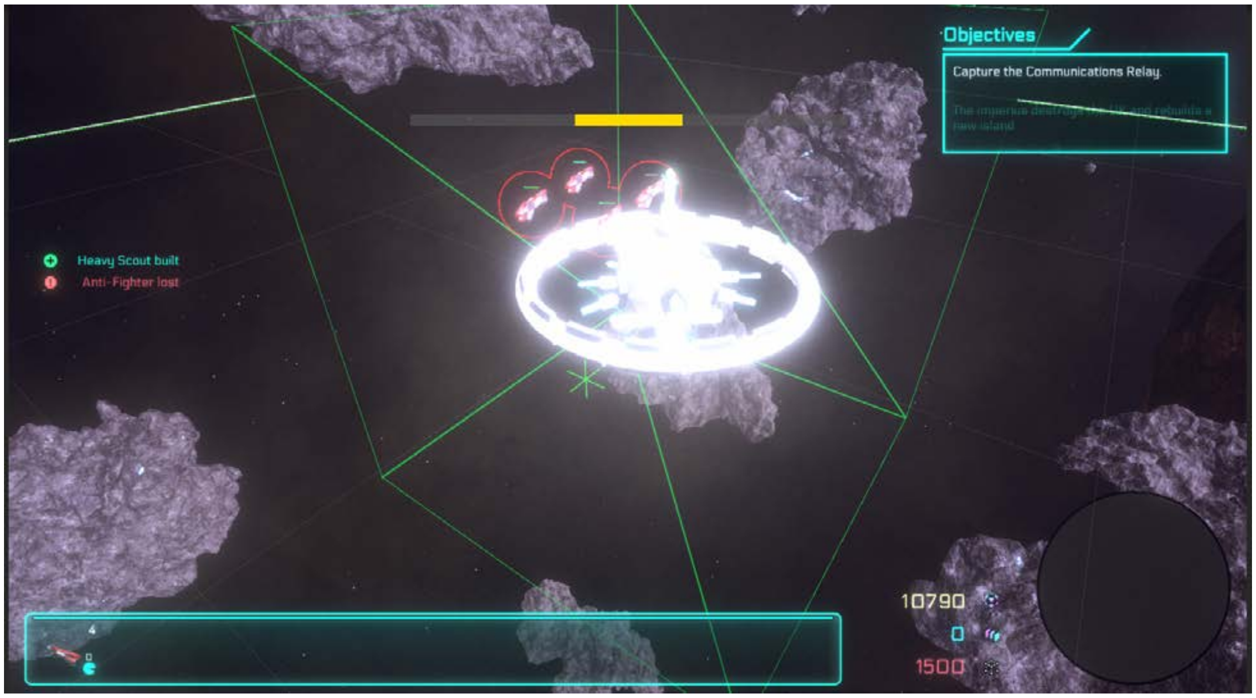


Figure 80: Capture Mechanic

## 9.3 Übermensch

Übermensch was designed around large-scale engagements and a massive battlefield. The core experience of this level is the utilization of every ship the player has at their disposal. The first hurdle the player must overcome is a barrier that destroys all ships greater than a certain size, so the player must destroy the shield generator using these smaller ships. This is so that when the reinforcements of larger ships arrive to combat the divinity reinforcements also on the way, the Imperius ships won't be blocked by the forcefield. On the other hand, Divinity ships, no matter the size, can move freely in and out of the forcefield and even fire out from it. Once the shield generator falls, the player must destroy the shipyards that were within the forcefield and are constantly producing units. It is a race against the clock that utilizes large-scale fleet engagements while also employing micro strategy through the management of groups of smaller ships. Because of the time limit this somewhat prevents the player from just mass-producing smaller units and forces the player to use what they have to achieve a larger goal.

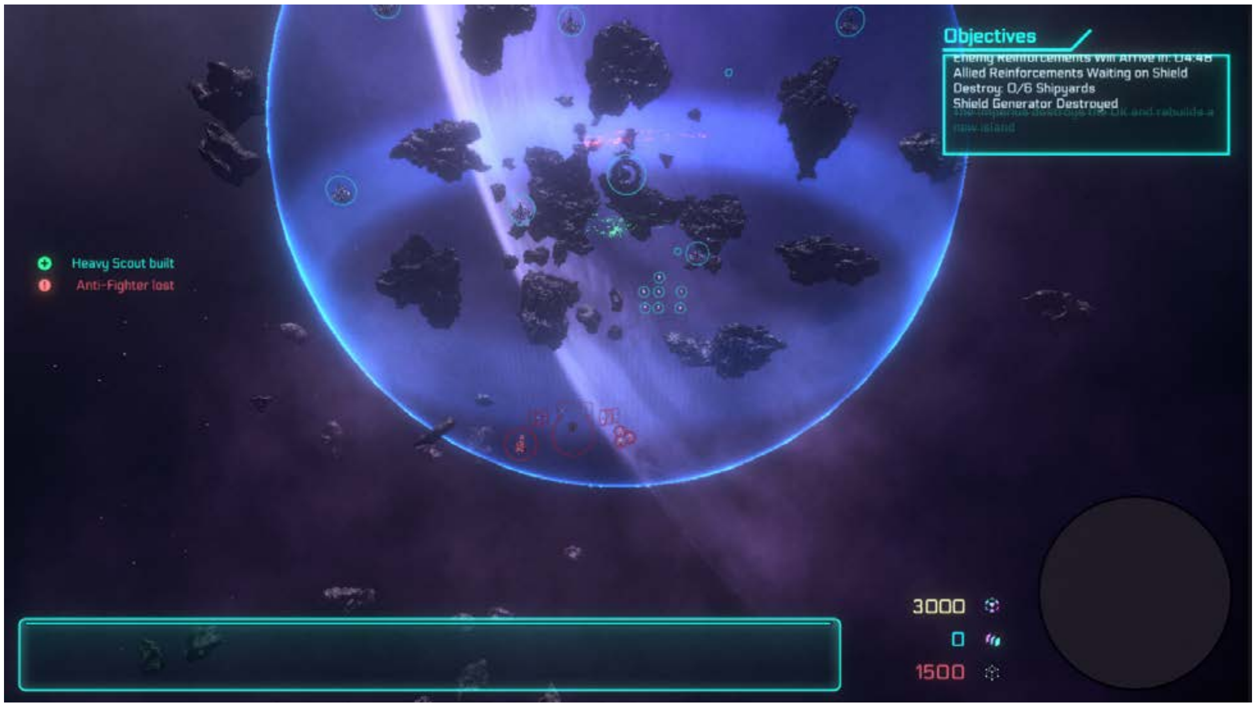

Objectives
Enemy Reinforcements Will Arrive in: 04:48
Allied Reinforcements Waiting on Shield
Destroy: 0/6 Shipyards
Shield Generator Destroyed
Heavy Scout built
Anti-Fighter lost
3000
0
1500

Figure 81: Wide shot of Übermensch

## 9.4 Perspectivism

Perspectivism was designed to be a step up in difficulty and progression from the mechanics of Butterfly Effect, as you are meant to survive and keep an objective alive for a certain amount of time. Perspectivism also played with the idea of how interactive we can make the level environments. The level narratively takes place in the same area as Übermensch so for continuity's sake, we reused the map from Übermensch, and so this meant that while designing Übermensch we also had to consider the needs of Perspectivism. This was achieved by essentially swapping the roles of the player and the enemies in this level. In Übermensch, the player was on the attack and the level was designed to give off that feeling through the placement of the asteroids, so in Perspectivism we placed the player's objective in the middle of this defensive formation and had the enemies attack them and make the player defend.

The level begins with the player needing to destroy several enemy ships, after which the ruined ship in the center of the level becomes active and the player must defend it. This adds an extra layer of complexity to the mechanics and design of Butterfly Effect, as even though in Butterfly Effect the player must keep some ships inside a set area to make progress if all the ships were wiped out from the zone as long as the player still has their base they can always make an effort to retake that point. In Perspectivism the goal is not to control a point but keep a stationary ship alive, so if a player's defending forces gets wiped out they don't have the luxury of building up a fleet to retake the point as the longer they take to defend the slimmer their chances of succeeding in the mission. Once the time is up the immobile ship will use its ability to launch a nuke and clear the way for the player to escape.

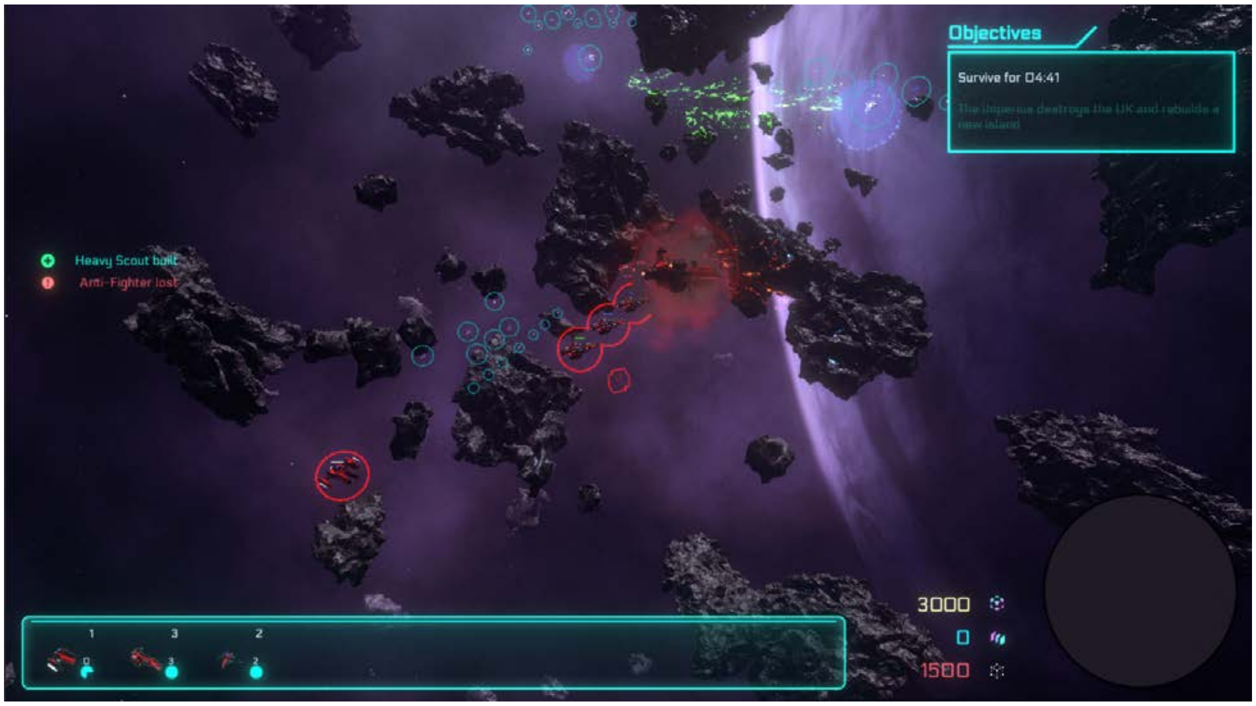


Figure 82: Defending the Derelict Ship

## 9.5 Stiletto

Stiletto was our stealth mission, one that took the micro strategy emphasized at the beginning of Übermensch and expanded upon it. The decision was made early on to cut off the player's means of producing units and buildings, taking inspiration from similar missions in other games. The next challenge was to make a stealth mission in a real-time strategy game that employed no fog of war, the solution we decided was to remove the shields from the player's ships. This made every engagement that much riskier and stacked against the player, encouraging them to find alternate solutions, such as stealth and positioning. For the first portion of this level, the player must destroy three enemy bases in an asteroid field that is dotted with turrets. If the player moves their ships recklessly then they will be torn apart by the high damage turrets. To counteract this the player was given access to plenty of Noctis stealth ships which can go invisible. The goal is to have the player carefully move their ships around the asteroids to take cover from the turrets and use the invisibility ability to their advantage.

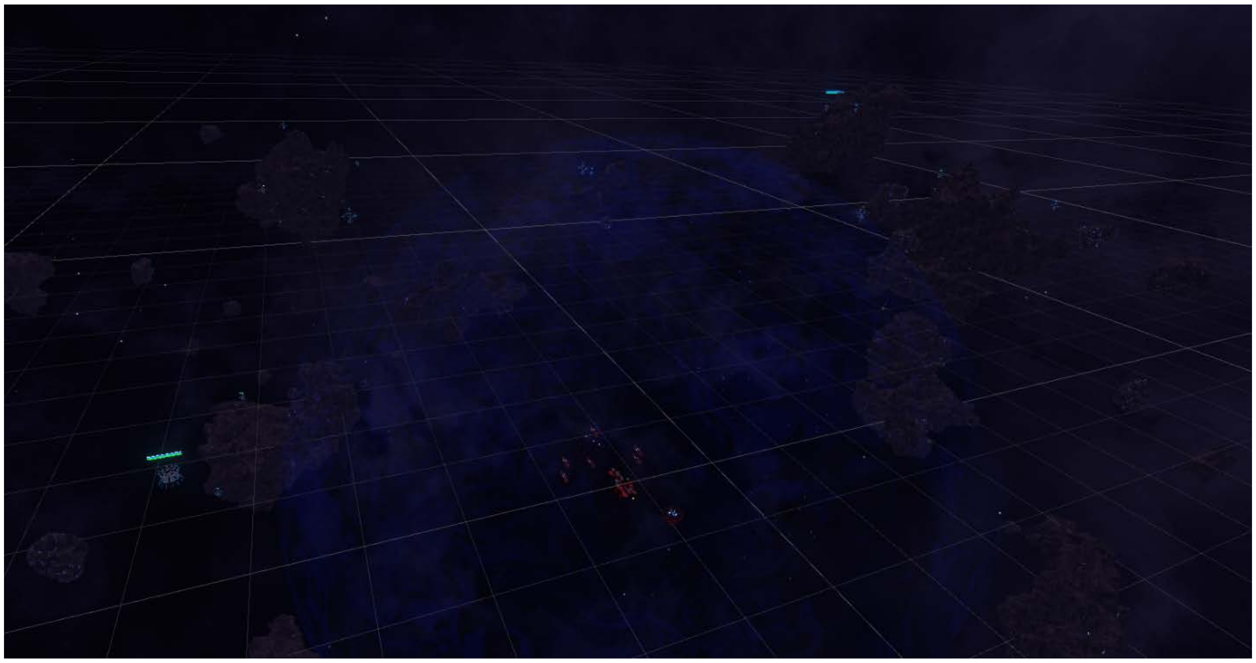


Figure 83: Stiletto Level

Once the three bases are destroyed the player must then successfully hold a point in these extreme conditions as enemy ships warp in and the player is given access to their means of production. This then tests their game knowledge as they are put at an extreme disadvantage, more so than similar survival and control levels such as Butterfly Effect and Perspectivism. To set the mood, deep blues, and reduced lighting are used to create a night scene in space. The bright gold, purple, and light blues of the divinity contrast with the background to make them stand out as targets.

## 9.6 Fractured

With Fractured, we wanted to sell the idea that this was the final mission and drive the twist ending home, and we felt that the most effective way of doing so was to have an endless horde mission that will not stop until the player dies. The mission takes place in an entirely open area, meaning that a player can no longer use the asteroids as a means of cover, but instead, their only defense now is the ships they can produce. In the distance, the player can see an asteroid field and beyond that another battle being waged over a planet. This is to help sell the narrative with as much visual feedback as possible, showcasing the current events in the story. The coloration of this level was also fairly important, as the red lighting and fog evoke a sense of finality and violence in addition to being the color of the player's and the Imperius's ships tying into the themes of the story occurring throughout this level.

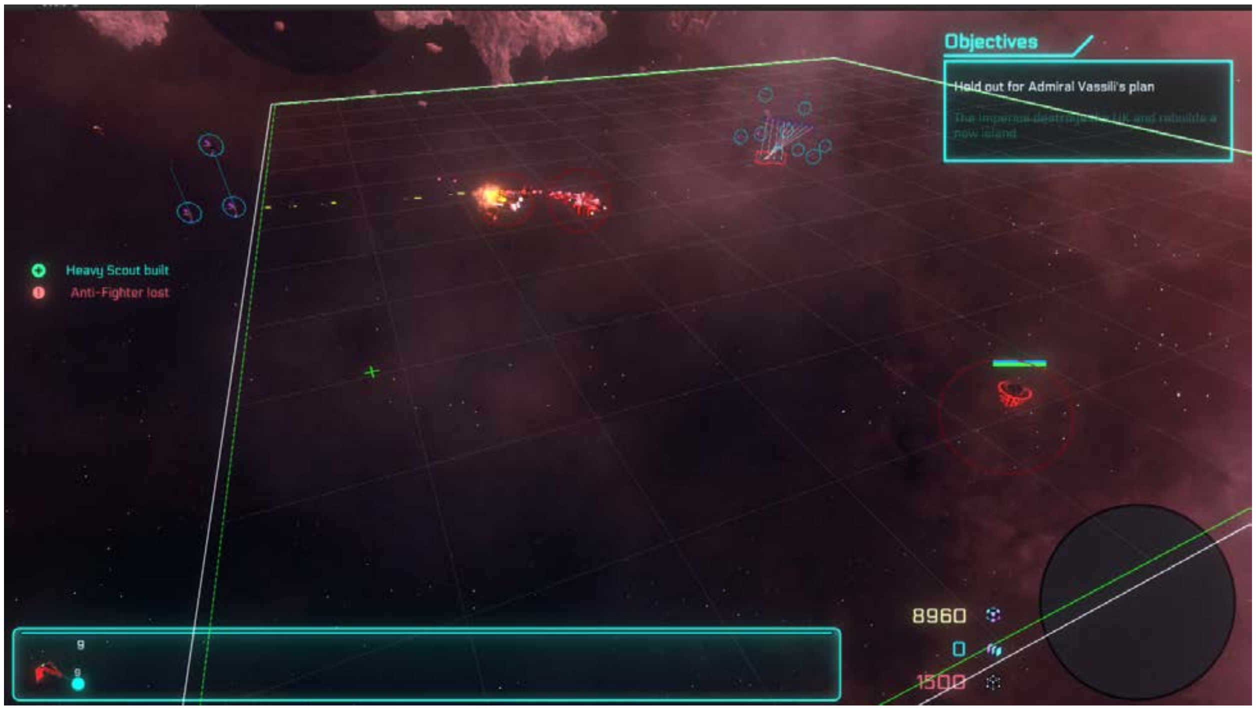


Figure 84: Can Only Use Your Ships as Defense

# 10. Narrative Design

When designing the narrative for the Imperius, our team drew inspiration from a variety of sources, primarily focusing on the dangers of blind loyalty and nationalism. However, as we delved deeper into the narrative design process, we encountered several challenges that needed to be addressed to effectively communicate our ideas in a way that would captivate the audience.

One of the main challenges we faced was the fact that most of the scenes in the narrative are expressed audibly, rather than shown on the screen. This made the concept of "showing and not telling" extremely difficult to achieve. It was important for us to find ways to convey the emotions and motivations of the characters through dialogue and sound effects, without relying too heavily on visual cues.

## 10.1 Introductory Cutscene

Marshal Sevda Severin is one of the main characters in the game's story, and the introductory cutscene is used to introduce her character and the game's setting and plot. The visuals in the cutscene (Figure 85) tell the story, and Sevda's struggle in the Divinity war is depicted. Her goal is to end the war, but later we find out that her efforts are in vain as her government is trying to stop her. The cutscene sets the stage for the narrative, and it is clear that the story will involve political intrigue and deception.

Figure 85: Concept Art for Introductory Cutscene

## 10.2 Pax Imperia

At the beginning of the Divinity Resurrection in IA 207, the player is tasked by Sevda to uncover the culprit behind the bombing of a peace rally. While searching for clues, players will experience flashbacks of the final battle of the Divinity War, which resulted in the Imperius winning. Throughout the game, players will also meet several key characters such as Vassili, Varus, Rudiger, and Jacqueline, who will provide additional information about the game's lore. As the investigation progresses, players will uncover that the Divinity has somehow obtained a significant advantage, and this kicks the investigation into overdrive.

## 10.3 Butterfly Effect

In this section of the story, the third fleet of the Imperius is tasked with tracking down the Divinity's activity toward a communications tower. Upon arriving at the location, they engage in battle with the surrounding Divinity forces and emerge victorious. The team then boards the tower and a new character, Colonel Minsheng, tries to extract valuable information from it. However, as they do so, more and more Divinity forces attack, putting their lives in danger. Amid the chaos, Osira Aryss, a powerful member of a new faction called the Insurgent Coalition arrives and confronts Vassili and Erwin. The two barely manage to escape with their lives after a fierce showdown, but unfortunately, Minsheng does not make it out alive. The events in this section of the story highlight the ongoing conflict between the Imperius and the Divinity and the high stakes involved in their struggle for power.

## 10.4 Übermensch

Admirals Erwin, Vassili, and Rudiger ally to confront the formidable coalition of the Divinity and Insurgent Alliance. Their objective is to launch an attack on their main base, hoping to deal a decisive blow to their opponents. During the attack, Sevda and Osira engage in a heated verbal exchange, bringing their history to light. Osira is revealed to have worked with Sevda before during a tragedy that destroyed the majority of the planetside forces. The tactical confrontation proves to be just as intense and results in heavy losses for both sides.

## 10.5 Perspectivism

Admiral Erwin is tasked with cleaning up stragglers from the previous battle and gaining a foothold in the area. The entire force is surrounded, Rudiger and Vassili take heavy losses, and Rudiger sacrifices himself to save the rest of the fleets. This is also a massive power grab for Vassili, as he puts you (his subordinate) in charge of Rudiger's fleet, which contains nuclear weapons to be used by him in the final mission to commit genocide.

## 10.6 Stiletto

We open with a black screen, as Osira tries to convince Sevda to help her prevent the death of everyone on the insurgency's planet of operations, but to no avail. Vassili leads a feint to divert forces while Erwin takes out a triangulation network. He gives you several new stealth ships to make your job easier. Halfway through destroying it, they locate Osira, who is speaking to the galactic community. She publicly accuses him before being captured by Iria, a commando under orders from Vassili. The station explodes, cutting off the transmission which appears to be a plea for help.

## 10.7 Fracture

We open the scene with a black screen of Osira's interrogation. It's insinuated that she's being tortured for the location of their home planet. You are tasked with leading a feint so Vassili can bombard the planet. Sevda realizes that Vassili has kept information from her and tries to call off the operation. He refuses. Osira helps break into your comms, and she broadcasts the order to everyone. Vassili orders them to keep fighting. As commanders start to panic, Varus turns on you and tries to shoot anyone who tries to leave. Chaos ensues as infighting destroys you and your fleet. He will then leave you to die as you see from a shattered viewport the planet being nuked to oblivion.

## 10.8 Ending Cutscene

Kaicus is speaking over the visuals, making a victory speech. He is telling us one story, while the visuals (Figure 86) are telling us another, interrupted by Osira's broadcasts of the truth of what happened. He talks and postures over the unfairness and incredulity of a surprise attack, but the reality is he engineered and planned the entire thing as an excuse to capture more territory. Erwin watches this from his escape pod and then kills himself once he realizes he was a tool for genocide. This dark ending is significantly more poetic, impactful, and realistic since most of the time individuals in power are not taken down easily. A happy ending requires a brutal, slow climb to victory, a goal you have been consistently working against in the entire campaign.

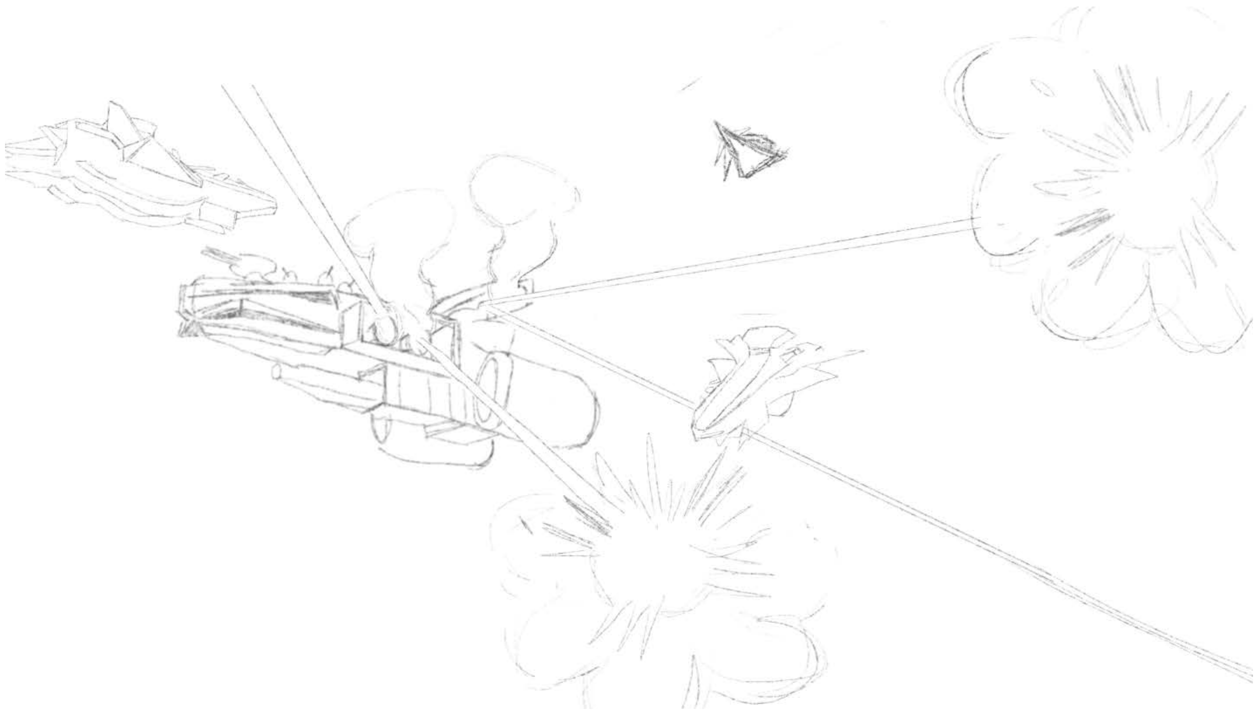


**Figure 8686: Concept Art of the Ending Cutscene**

## 11. Playtesting

We conducted several playtesting sessions throughout the year and asked players to provide feedback on various aspects of the game. As shown in Figure 87, we created multiple posters to advertise our playtesting sessions. We hung them up in various buildings around campus. After each person finished playing, we had them fill out a feedback form and give their thoughts on the game's difficulty, graphics, UI, music, sound effects, and level design. In addition, we requested that they report any bugs they found and offer suggestions for how we could improve the game. We utilized the feedback received from playtesting to make iterative improvements to the game, ultimately enhancing the player experience.

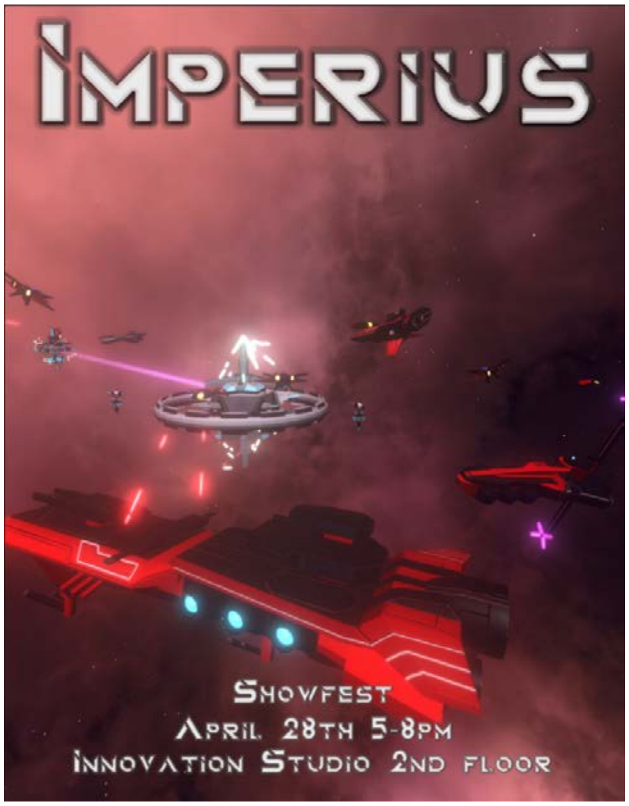


**Figure 8787: Showfest Promotional Material**

During the playtesting sessions, we encountered both new and experienced players with RTS games. The inexperienced players took considerably longer to grasp the game mechanics and beat the demo level. Conversely, most of the experienced players were able to complete the level swiftly, and with ease. We took note of where the newer players struggled and amended the demo level and tutorial accordingly. For instance, we removed the shield from the level Übermensch because newer players had difficulty realizing that they had to send smaller ships to destroy the shield generator before they could progress. Partway through the year, we also introduced an interactive tutorial that guides players through each available action in the game, such as the selection of ships and the construction of buildings. These two changes improved average player performance drastically.

Additionally, we learned from the players' strategies, noting that some built more factories than intended, generating an excessive amount of materials. To address this, we implemented a power mechanic that limits the number of materials you can obtain from factories based on the number of solar arrays you have.

# 12. Conclusion

In conclusion, Imperius is a 3D real-time strategy space game developed by a team of talented WPI seniors over the 2022-2023 academic year. The game boasts 24 unique ship units, a story campaign with 6 missions and 2 cutscenes, 30 minutes of music, 40 sound effects, and 4-player Steam integrated multiplayer. With the help of two other WPI students through independent study projects and around two dozen voice actors, the team was able to bring the game to life. Imperius is a testament to the culmination of several years of work in multiple disciplines, including Computer Science, Software Engineering, 3D modeling, SFX, VFX, Level Design, and more. The project showcases the team's skills, dedication, and passion for game development.