Major Qualifying Project

Michael French
Adam Ansel
Tom Hunt

Advisors: Britt Snyder, Keith Zizza

Worcester Polytechnic Institute

Table of Contents

Contents

Abstract

As 3-dimensional graphics for videogames became popular, a genre of games was born called “flying games.” Flying games typically had players control an airplane or spaceship and often involved combat with projectile weapons. Meanwhile, many fantasy works contain characters who ride a flying mount, such as a pegasus, yet fight with a melee weapon, such as a sword. While many videogames include such characters, few explore what combat between them looks like. In order to explore this combat, we created *Aeroknights: Deathmatch*, a video game in which players ride flying bikes and attack each other using mainly melee combat instead of projectile combat. The game played like a very up-close and personal version of the flying games seen before, and create a unique and exciting way for players to interact.

Introduction

Inspiration

In the 1990s, as 3-dimensional graphics gained popularity, the flight-sim videogame genre also gained popularity. These games generally had players controlling a space-ship or airplane, and mostly involved dogfighting other similar crafts. In popular flight-sims such as *Star Wars X-wing*[1], *Freespace 2*[2], and *Wing Commander*[3], players would utilize weapons such as lasers and missiles to combat opponents and had full control of their ship's pitch, yaw, and roll axis. The appeal of these games was largely considered to be the immersion they provided as they allowed players to take on the role of a pilot seen in movies such as *Star Wars*[4] and *Independance Day*[5]. Flight-sim battles were often about maneuvering as players wrestled to put their opponent in their sights and fire.

A player engages a Star-Destroyer in Star Wars: X-Wing

[1] Star Wars: X-Wing: LucasArts. LucasArts. 1993. Video Game
[2] Freespace 2: Volition, Inc.. Interplay Entertainment. 1999. Video Game
[3] Wing Commander: Origin Systems, Inc.. Origin Systems, Inc. 1990. Video Game
[4] Star Wars Episode IV: A New Hope. Dir. Lucas, George. Twentieth Century Fox, 1977. Film.
[5] Independence Day. Dir. Roland Emmerich. 20th Century Fox, 1996. Film.

Meanwhile, fantasy works such as *World of Warcraft*[6] and *Fire Emblem*[7] contained characters that would ride on a mount which could fly. Following the fantasy genre, these characters would often be seen wielding weapons such as swords, spears, and axes. While these characters and concepts exist, combat between them is scarcely found in written works, films, and videogame cutscenes. As such, few videogames have attempted to tackle the idea of flying characters with primarily melee weapons as a central game mechanic.

A Pegasus Rider from Fire Emblem wielding a spear

One game that did attempt to have melee combat while flying was called *The Last Phoenix*[8], which was released on indiedb.com by Barbossal on January 25th, 2013. In this game, players assumed the role of a phoenix who fought against crows and other aerial and ground-based enemies. Players could utilize two kinds of attacks: a ramming attack which would give a short speed burst and damage all enemies in the player's way, and a grappling attack which would grab enemies, allowing the player to peck them to death. The game used keyboard-and-mouse controls, with the mouse used to aim attacks and steer the character, and the keyboard used to control speed and other miscellaneous functions. While *The Last Phoenix*

[6] World of Warcraft: Blizzard Entertainment. Blizzard Entertainment. 2004. Video Game
[7] Fire Emblem: Shadow Dragon and the Blade of Light: Intelligent Systems. Nintendo. 1990. Video Game
[8] The Last Phoenix: BearInMind games. BearInMind games. 2013. Video Game

was a great take on flying melee combat, the main character was a single entity instead of a mount and rider, as seen on the examples above.

A player attacks a target in melee in The Last Phoenix

The goal of *Aeroknights: Deathmatch* is to create a compelling combat system which primarily involves player riding a flying mount (or vehicle) and attacking with melee weapons. Unlike *The Last Phoenix*, our game will feature characters who are mounted on flying vehicles, and will thus have combat that reflects the disconnect between weapon and vehicle. As explained above, there is a general lack of games that involve this concept and most flying games use projectile weapon combat. In this project, we endeavor to explore the implications of limiting players to melee weapons when combating each other off-ground. It is hoped that *Aeroknights: Deathmatch* can fill the niche for a flying melee combat and serve as a possible example for future games to learn from.

Implications of Melee-Centric Combat In Flying Games

In a videogame, a melee weapon is typically identified as a weapon used by players to fight at close range without projectiles, such as baseball bats, swords, or pikes. Projectile weapons are weapons that use a projectile as their primary form of attack, such as a gun, bow-and-arrow, or slingshot. Typically, projectile weapons can attack from a much greater range than melee weapons. Most existing flying video games use projectile weapon combat, usually with machine-gun type weapons and homing missiles. We feel that, in many cases, projectile weapons allow players to shoot down opponents without the opponent being able to react. We thus theorize that using melee combat in a flying game will force players to interact with each other more than they would with projectile combat.

A player shoots down an enemy from afar in Pacific Fighters

In a versus game, we hypothesize that players make the most game decisions when they are in range of their opponent and will tend to make more drastic movement decisions the closer they are to their opponent. In a flying game with melee weapons, players must fly close to their opponents in order to attack. This means that, compared to ranged flying games, players

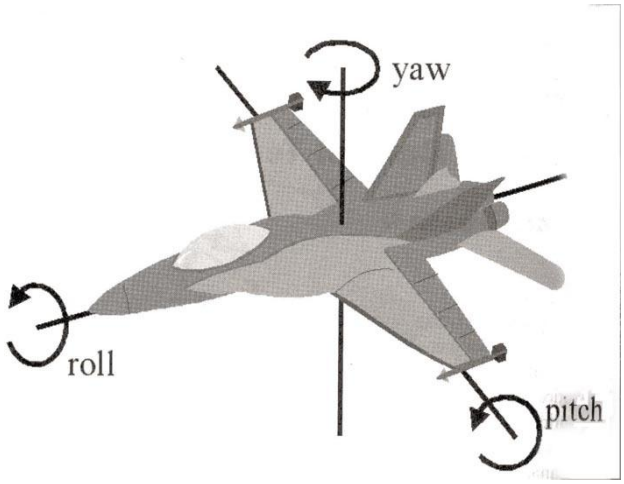
would need to fly much more acrobatically when combatting their opponent. Specifically, when an opponent gets too close to a player, that player could do several things to evade: dive down, pull up, turn sideways, fly in a spiral, or even fly in a loop-de-loop.

With so many decisions to make while actively fighting their opponent, players would also need to strategize how they are going to initiate their attack. Assuming both players fly at the same speed and maneuverability, attacking from behind may not be as beneficial as it would with projectile combat. Doing so would require the player to catch up to his opponent in order to actually deal damage, which could be cumbersome when compared to attacking from the front or flank. Instead, a better strategy may be to attack the opponent from the flank in order to limit their evasive options.

Design

Controls

When deciding our control scheme, there were three major controls to consider: the player must be able to steer their ship, manage their speed, and control their weapon. The optimal combination of these controls would have as few inputs as possible in order to keep the game intuitive and allow for more functionalities. At the same time, we also needed to make sure players would be able to execute as many evasive and aggressive maneuvers as possible.



The three major axes of a flying vehicle

Since this is a flying game, steering the ship would mean control over pitch and yaw at the very least, with the possibility of roll. We came up with three control schemes. The first scheme involved direct control over the ships pitch and yaw with the ship always oriented right-side-up to remove the need for roll control, as seen with the Banshees in *Halo: Combat Evolved*[9]. We decided this implementation would have meant that the ships were most maneuverable when their pitch was at 0 degrees, which would limit evasive options. In an

[9] Halo: Combat Evolved: Bungie. Microsoft. 2001. Video Game

attempt to fix this, we tried making the pitch fixed as well, making it so the ship was always parallel with the ground, then replacing the pitch controls with “altitude” controls, as seen in *StarFox 64*[10]. This, however, created a very arcade-like experience that seemed to limit the excitement of flight. We ultimately decided to give players direct control over pitch, yaw, and roll; using the mouse for pitch and yaw, and the keyboard for roll. This control scheme would provide the highest level of control needed to make the game engaging.

For speed management, there were two possibilities: either speed was fixed with dedicated “go fast” and “go slow” buttons, or speed was variable with dedicated “speed up” and “slow down” buttons. The former option would offer quicker movements, while the latter option would allow for more controlled flight. We decided to use variable speed.

We boiled the issue of weapon control down to two possibilities. We could A) have the weapon’s area of attack in a fixed area relative to the player, or B) have the player aim the weapon with the mouse input. Option A would require the player to be oriented to their opponent at a specific angle in order to hit, while option B would provide a wider range of attack options. Option B would also make the weapon’s aim consistent with the player’s steering. Considering the proximity players must be in in order to attack, we decided to use option B to allow for more attack variety.

When considering the view perspective, there were two possibilities to consider: first-person and third-person view[11]. A game using first-person view has the player viewing the world through the eyes of their character. First-person games will often have the player-character’s arms animated to convey certain actions, like attacking and switching weapons. A game using third-person view, however, has the player viewing the world through an invisible camera facing towards the player-character, usually from behind. First-person games tend to be more

---

[10] Starfox 64: Nintendo EAD. Nintendo. 1997. Video Game

[11] "What Are the Differences Between First-Person Shooter and Third-Person Shooter Games?" EBay. N.p., n.d. Web. 05 Mar. 2015. <http://www.ebay.com/gds/What-Are-the-Differences-Between-First-Person-Shooter-and-Third-Person-Shooter-Games-/10000000177589743/g.html>.

immersive than third-person games, as the game's gestures and actions are addressed directly to the player's point of view. Third-person games tend to allow more spatial awareness than first-person games, since the view allows the player to see areas adjacent to and behind their character. For *Aeroknights: Deathmatch*, we wanted players to feel completely aware of their surroundings as they navigated obstacles and evaded enemy players. We thus decided to use a third-person perspective

Weapons, Abilities, and Pickups

When considering the main weapon, we boiled the weapon types down to two distinct options: lance-like and sword-like. A lance-like weapon would extend in front of the player and require players face their opponent in order to hit them. A sword-like weapon would attack from the player's flanks and require players to be side-by-side in order to hit each other. When considering these options, we had to think about how the flight controls would influence players' attack strategies. We figured the forward facing camera would encourage players to fly straight for their targets, which is best done with the lance-like option. Lance-like weapons would also create a vulnerability behind players, where the sword-like option would only put both players in danger when one attacks the other. Based on these hypotheses, it was obvious the lance-like option was more suitable.

During early play-testing, we observed a tendency for players to get locked in a chase where one is trying to catch up and do damage, while the other is trying to evade their aggressor. This chase proved to be difficult to break, so we added a few mechanics to help both the aggressor and the defender. We added a speed boost that players can use at the cost of maneuverability and a resource called Fuel. With this mechanic, aggressors can lead their target and speed into their opponent's path and hit. We also added mines that evaders can use to dissuade chasers from following them. If a player ran into one of these mines, it would explode and damage them. The number of mines available would be limited.

Map

When designing our map, we needed to strike a balance between density and openness. If the map were open (devoid of features), every player would be able to see every other player at all times and there would be no obstacles to fly around. This would not only make sneak attacks and strategic positioning difficult, but also make it more difficult for players to lose pursuing opponents. If the map had too many obstacles in it, however, then navigation would become difficult. In order to have the right amount of obstacles, we decided to make the map in a city-like location with skyscrapers. These skyscrapers could be strategically placed to create interesting paths to fly through.

Left: a player attempts to navigate a very dense map, Right: a player has nowhere to go while being attacked in highly sparse map.

We wanted to encourage players to take risks when flying in order to boost excitement. We figured that, while not fighting, players would have little reason to attempt to fly through obstacles. We thus decided to add pickups to the game. A pickup is a floating object that, when touched, gives the player some sort of bonus. We came up with four bonuses a pickup could yield: health replenishment, fuel replenishment, mine replenishment, and an energy projectile. Placing these pickups in certain areas, obstacles or not, would encourage players to go to said areas and increase variation in combat encounters.

Player Heads-Up Display

The HUD in StarFox takes up little screen space, allowing the player to see more of his surroundings

We decided that, in order to keep track of things like health and pickup resources, the player would need to have a heads-up display (HUD). The most important piece of information the HUD would need to convey is the player's health, so the heathbar would be the biggest part. Additionally, the HUD should also show the player how many resources, such as fuel and number of mines, he has left. The HUD would need to display this information in a compact manner so as to not reduce the player's spatial awareness.

Art

Inspiration

The overall aesthetic of *Aeroknights: Deathmatch* traces its inspiration to a variety of sources, both founded in reality and science fiction. One could consider the art style of our game to be "dieselpunk," in that it combines "the aesthetics of diesel-based technology influenced by the interwar period to the 1950s with futurist postmodern technology."[12] The decision to take the art style in this direction came about with the creation of the Aerobike vehicle. The remaining elements of the game are designed to be consistent with the theme set forth with the Aerobike. The setting of *Aeroknight: Deathmatch's* game environment is inspired by the sci-fi bio domes prevalent in science fiction movies and literature. One of the initial inspirational elements for our environment is Cloud City from *Star Wars*. The tall, sculpted buildings provide an unmistakably futuristic look.

The design of the Aerobike vehicle itself is partially inspired by mid-21-century jets, such as the F-86 Sabre. Originally, it had been our plan to situate a single intake right at the nose of the vehicle, but later consideration led us to replace it with a grille and situate the intakes on each side instead. The silver coloration was also inspired by the F-86. Certain aspects of the Aerobike can be traced to motorcycle design, such as the handlebars with brake handles.

[12] "Dieselpunk." Wikipedia. Wikimedia Foundation, n.d. Web. 06 Mar. 2015.

The Aerobike (right) and its inspiration (left)

The development of the second vehicle began late in the project. Since our overall art style had already been established, it was both easier and more difficult to imagine how the new vehicle would look physically – easier in that we already knew what sort of look to aim for, and more difficult in that there was less creative freedom for appearance. With these constraints in place, the new vehicle turned out as sort of a cross between the original Aerobike and the *Naboo Starfighter*. In order to differentiate the design from the preexisting Aerobike, we referenced the steering setup of a snowmobile (handlebars on a central rotating pivot) rather than rigid handlebar setup (as on the original Aerobike) to more clearly distinguish between the two.

The Turbobike (right) and its inspiration (left)

A secondary influence for the Turbobike was the design of the P-38 Lightning. The influence of the P-38 is apparent in the twin-tail of the Turbobike. Additionally, the commonly silver color scheme of the Lightning factored into our decision to give the Turbobike a reflective silver finish as well.

A P-38J, which also influenced the Turbobike

The process of envisioning, designing, and constructing characters for Aeroknights: Deathmatch took several steps and resulted in the complete re-working of the characters from the initial design concept. Originally, we had planned on producing several biker-style human characters with leather jackets and associated paraphernalia, but this concept devolved into something more in-line with the absurd overall premise of the game: robots vs. robots.

An advantage of using a robot as a character model is the straightforward rigging in comparison to an organic humanoid model. To rig the character, each segment of the body needs to be parented to its corresponding bone.

The Robot which rides the bikes

Decisions

One of the decisions we made during our design process was to switch the semi-realistic appearance of our game with a cartoony appearance instead. The use of a toon shader is also beneficial to the overall game aesthetic, as it provides a consistency between the more fantastical elements of the game (robots riding jet bikes, for instance) and the aspects of the game that would not look too out of place in real-life modern urban architecture. In games that use a visually realistic art style, the player may expect a gameplay experience more founded in the real world. In our case, we did not want to promote this possible association.

This stylistic transformation took some manipulation of the existing toon shaders and extensive online searching to preserve certain aspects of the game's appearance, but the result is to our satisfaction.

Asset Production and Implementation

City Buildings

In order to produce the buildings for our virtual city, we modelled each building separately and used a particle system within Blender 3D to arrange the buildings somewhat randomly. However, this did not lead to a layout that was compatible with the central “Rings” in our game environment, so we ended up modifying the arrangement of the buildings to accommodate the central garden rings. The entire city layout was imported into Unity as an FBX, allowing us to keep everything grouped together as a prefab. Sub-objects can still be rearranged by selecting them within the city prefab - this allows us to reposition individual buildings.

The City in which the game takes place

Skybox

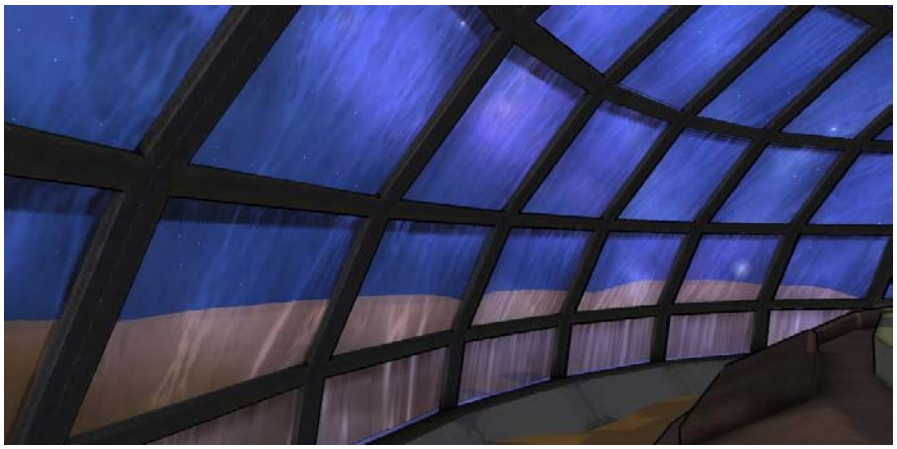
The space skybox for *Aeroknights: Deathmatch* was painted by hand on the interior of a sphere in Blender, then projected onto the outside faces of a cube. These faces formed the

base layers of the skybox. The next step in the skybox construction was to generate random stars in Spacescape and overlay these onto the painted nebulae within Photoshop.

A piece of the skybox texture

Dome

The dome around our game environment is composed of two sections. The inner part of the dome is made up of the support structure and a transparent layer for the window panes. Directly outside this inner dome is an outer dome with a grunge texture applied. The reason we had to set up the dome this way was to compensate for the lack of a shader that supports transparency, reflectivity, and a diffuse texture. The outcome of this process provided the results we were looking for.



The composite dome material

User Interface

The user interface or Aeroknights: Deathmatch is designed with a grungy industrial aesthetic in mind. Stained metal and frequent use of bolts on the menu panels reinforce this motif. Rolling over the buttons produces both a highlight effect and a sound to let the user know that the button has received their input. In order to create the various faux-metal panels for the game UI, we located stained or otherwise dirty metal textures online and modified them in Photoshop, such as through the addition of edge bevels. Bolt images are overlaid on corners to finish the look.

Music

We chose the soundtrack for our game to encourage fast-paced action. To this end, we have made use of two tracks from the user ScorcherEdits on YouTube - each in the same key but with a different tempo. Circumstances within the game environment trigger transitions between the two tracks. Specifically, the slower-paced track plays continuously until the player passes within a certain distance from an enemy player, at which point the first track fades into the second faster paced “combat” loop. This second track continues to loop as long as the player remains under the set distance from an enemy. Upon exiting this range, the music

crossfades back to the first track. In this way, *Aeroknights: Deathmatch* takes into account the action on screen for each character to dynamically change the game soundtrack.

Development

Game Engine

At the start of development, we were had to figure out how we going to develop *Aeroknights: Deathmatch*. Our team had two high level options: develop the game from scratch or utilize a game engine. Since our ultimate goal was to explore a certain mechanic in depth, we decided that developing the game from scratch was unnecessary. We thus chose to use a familiar and well supported game engine called Unity[13]. Unity offers a robust set of core features including built-in rendering, collision detection, and physics. Unity also allows for all types of art assets to be used including 3D models, textures, materials, and particle effects. This ultimately gave our artist free range within his capabilities to make and implement anything he wanted. Unity also features built-in networking capabilities for online multiplayer games, which proved useful when we decided to make *AeroKnights: Deathmatch* multiplayer. The decision to use Unity ultimately allowed us to develop core game functions early, giving us time to play-test and tweak the game throughout development.

One feature of Unity we commonly used was the prefab[14]. Prefabs are objects which hold components and scripts, and are a way building objects that can be loaded into the game. The components of a prefab can include collision boundaries, mesh renderers, scripts, and even particle effects. Prefabs can also have child prefabs, which will share collision detection and move with the parent prefab. Finally, variables made public in scripts can be altered in Unity's GUI, making playtesting and variable tweaking much more streamlined and efficient. We

[13] Unity. Unity. N.p., n.d. Web. 01 Mar. 2015. <http://unity3d.com/>.
[14] "Prefabs." Unity. N.p., n.d. Web. 05 Mar. 2015. <http://docs.unity3d.com/Manual/Prefabs.html>.

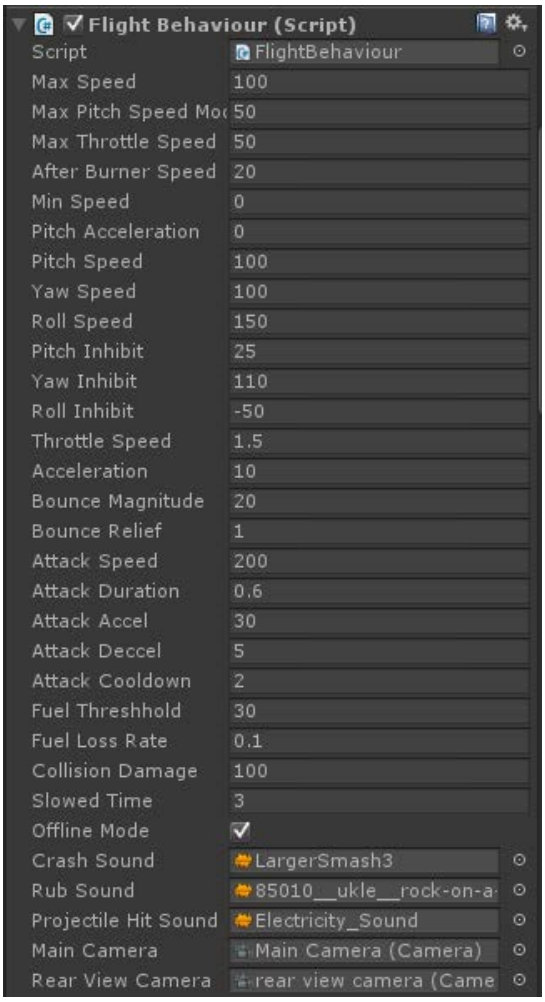
used prefabs to build every major object in the game, including the player character, network manager, pickups, and spawn-points.

Player Object

The first object we needed to create was the player character, which would serve as the object which the player controls as the game is played. The player character would need to take in keyboard and mouse input, and produce movement and control of game mechanics. We decided early that different mechanics would be handled by different scripts, which would all be loaded on the same prefab. For instance, the PlayerInput script would read keyboard and mouse input, which the FlightBehaviour script would use to handle flight mechanics like speed and turning.

In order to allow these scripts to gather input from a single source, we created the PlayerInput script to manage all keyboard and mouse input. For most inputs, we simply check if a key was pressed, but some inputs are more complicated. In order to manage mouse input for steering, we utilized the mouse cursor position relative to the middle of the screen to report an analog value between -1 and 1 for pitch and yaw. When the mouse cursor is in the bottom-left corner of the screen, the script reports the steering inputs to be -1 for pitch, and -1 for yaw. When the mouse is in the top-right corner of the screen, the script reports the steering inputs to be 1 for pitch and 1 for yaw. These inputs would be used to steer the player and aim the player's weapon.

For the player's flight behavior, we we created the FlightBehaviour script, which manages the player's speed, acceleration, roll, pitch, and yaw. We made each of these variables, as well as the variables affecting them, public in order to be able to tweak them from Unity's GUI. This allowed us to use the same script for both the Aerobike and the Turbobike. To make each bike handle differently, we simply modified the bikes' speed and turning variables in the Unity GUI. In order to allow player control, the FlightBehaviour script gets input values from the PlayerInput script. The throttle, for instance, gives the player precise control over their speed, and is increased or decreased depending on PlayerInput's getThrottleInput function.

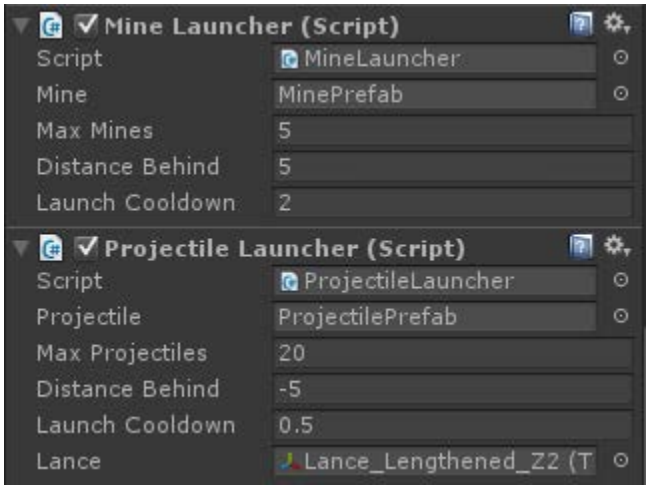


The variables for FlightBehaviour were editable in Unity's GUI

In addition to speed and turn control, FlightBehaviour also has an afterburner, which provides a steady speed boost, and a dash attack, which quickly launches the player forward. The afterburner is activated when the afterburner button, mapped as the 'shift' button by the PlayerInput script, is held down by the player. The dash attack is activated when the the player clicks the left mouse button a dash attack happens when the player presses the attack button, which is mapped as the left mouse button. The dash attack increases the speed of the bike for a split second, allowing the player a burst of speed needed to attack his opponent.

The main variables of the PlayerHealthManager

AeroKnights is a fighting game at heart, so each player needs to have a certain amount of health. If the player's health reaches zero, the player dies and must respawn. The PlayerHealthManager script handles and processes the current health of the player. If the player gets hit, health is lost and on-screen effects are played to represent the damage. A variable set in the HealthManager script determines the bike's max health, and functions within the script to determine what to do to that variable. For example, within the script there is a function called loseHealth, and if loseHealth is called we can determine how much of the bike's health is lost anywhere else in the game. This allows us to decide what damages the player, be it a lance attack or environmental hazards.



The main variables for the Mine Launcher and Projectile Launcher

We decided to have a form of attack in addition to the player's dash attack. In *AeroKnights: Deathmatch*, the main attack is a dash attack with the lance the character is holding, which is handled in the FlightBehaviour script. For a secondary attack, we added mines, which are stationary objects that damage opponents when touched. The purpose of

mines is to give players who are being chased a way to deter their aggressors. In order to enable the launching of mines, we created the MineLauncher script. MineLauncher handles the number of mines the player has, and the method of launching them. To launch a mine, MineLauncher will place a mine object, which is a prefab, behind the player. The mine can only damage opponents, and will not harm the player who launches it.

In order to convey these mechanics to the player, we implemented a heads-up display (HUD) that would be visible while the player was in-game. At the top of the HUD, we created a health-bar, which would display the player's health as both a progress bar and a number. We also created a small panel that would display information about the player's current pickups, such as mines for the mine launcher and fuel for the afterburner. Finally, as an aesthetic addition, we put a light in the HUD which would glow bright red when the player's health was below a certain threshold.

Multiplayer

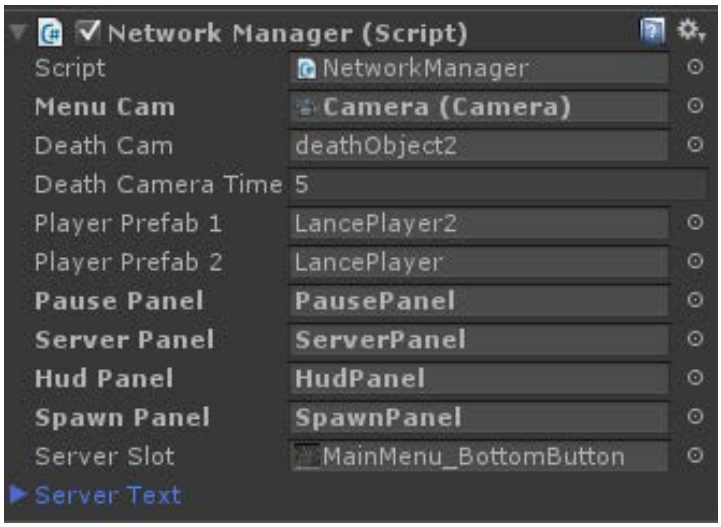
Early in development, we faced a difficult issue: how could we create an opponent that would adequately test the idea of melee combat in a flying game? In order to fully explore the concept, the players' opponents would need to adapt to the environment around them. If players adapted a new strategy, the players' opponent would also need to adapt to prevent the game from becoming stale. We considered two high-level options for the opponent: creating autonomous bots or making the game multiplayer. Doing both options was out of scope for the project, so we had to choose one.

Creating a bot would require a complex AI that would account for a large amount of minute details including its environment, positioning, speed, and position relative to other players or bots. It would also need a contingency for each possible situation, such as when it is in front of the player, behind the player, or above the player. As strategies for the game got fleshed out, the bot's programming would become more complex. Implementing multiplayer, on

the other hand, would merely require network programming that would allow multiple computers to play within the same space. We ultimately decided that, in order to adequately explore melee combat in a flying game, implementing multiplayer would be a far more efficient solution.

The Network Manager

In order to make our game multiplayer, we needed to create network manager that would handle both server and client data. To do this, we used Unity's Network class. For server creation, we used the function Network.InitializeServer, which does as the name implies. This function can be configured to use or ignore network address translation (NAT), making it possible to use it for creating both local area network servers (LAN) and online servers. Once created, if the server is not LAN, it is registered to Unity's master server, which allows clients to see a list of all servers currently available for our game. This allows players to host servers that others can to browse to and play on.



The Network Manager

In order to transfer information over the network, we used Unity's built in network view component. Network views are attached to prefabs that need to be synced on each player's computer. Each computer will have its own copy of any object with a network view that is

updated from said object's primary computer. We attached a network view to the player prefab and configured it to send only information from the FlightBehaviour script. In FlightBehaviour, we also wrote a check to see if the prefab belongs to the computer displaying it. If it does, FlightBehaviour would be controlled by that computer, otherwise, FlightBehaviour would update itself based on the information given by the source computer. Similar techniques were implemented with the pickups, mines, and projectiles.

By default, network views only send the object's position over the network. This would cause networked players to move in a very jittery pattern as the vehicle forward on every update. This is because of the server latency. In order to mitigate the jittery movement, we wrote our FlightBehaviour script to send the velocity of the vehicle along with its position. This made it so networked players seem to interpolate between positions as they are updated, and makes their movement much smoother.

Network views do not track player statistics such as health and pickup resources. This meant that, when one player damages another, we needed a way to tell all of the clients that a player has taken a certain amount of damage. To do this, we created a PlayerHealthManager script, which would track the player's health and communicate it through the network. In this script, made two functions which cause the player to take damage; takeDamage and takeDamageRPC. The takeDamage function modifies the local player's health, then does a Remote Procedure Call (RPC) on takeDamageRPC. This tells all of the other clients to call takeDamage on their version of the player who's supposed to take damage.

Spawning

Managing how players would spawn into the game was important. If a player spawns too close to a building, he could crash before being able to react. Additionally, if a player spawns too close to another player, one of them could gain an unexpected and random advantage. To avoid these situations, we created spawn-point prefabs and strategically placed them

throughout the map. Each spawn point was placed at an edge of the map away from other spawn points, above the buildings, and roughly facing the center of the map. This placement would give newly-spawned players space to orient themselves before jumping into battle, and the orientation would encourage players to fly to the center of the map where they are likely to find other players.

We developed an algorithm to determine the best spawn point for players to spawn at. When a player spawns, the network manager queries each spawn point for a priority. Each spawn point determines their priority by adding the distances from that spawn point to each player in the game. The network manager then chooses the spawn point with the highest priority score and instantiates the player prefab at the object's position and orientation. As a result, players will always spawn as far away from the action as possible.

nGUI

During development when discussing the user interface and how it should be implemented, the topic of nGUI[15] was brought to our attention. nGUI is a plugin for Unity that allows for powerful control over the user interface and HUD. Unity's default Graphic User Interface (GUI) functions are done pixel-by-pixel, which makes it difficult to create a GUI that looks consistent on different screen resolutions. NGUI allowed us to create UI elements such as buttons and progress bars within the scene that would resize properly depending on the resolution. This means we can edit exactly how the HUD and menus will look like without having mathematically define positions in code, making our GUI design process much more efficient. NGUI also allowed us to utilize our art assets in a way that can easily convey information to the player**.** By communicating with other scripts such as the PlayerHealthManager and

[15] "NGUI: Next-Gen UI Kit." Tasharen Entertainment. N.p., n.d. Web. 05 Mar. 2015. <http://www.tasharen.com/?page_id=140>.

FlightBehaviour scripts, nGUI can access information such as health and fuel. Accessing these variables can then be applied to nGUI elements to display current game information.

Art Pipeline

For our art pipeline, we had our artist create the assets using Maya, Blender, After Effects, and Photoshop. Maya and Blender were used to create the game's 3D models. Additionally, Texture Packer was used to compile image sequences for animated sprites into one texture atlas. A script within Unity was used to scroll through the images in the atlas to give the effect of continuous motion. After Effects was used to create the animated 2D assets, such as the title-screen background. Photoshop was used to create materials for the 3D models as well as the panels and buttons for the menus and player heads-up display.

Once the art assets were created, they were imported into our Unity project. Integration was mostly seamless as Unity supports a wide variety of file types for art assets. Many of the models used unique materials for different components - this allowed us to adjust the materials in Unity with distinct properties, such as reflectivity, self-illumination, etc. Some of the models also needed to be scaled once imported into Unity, as the scaling conversion between the 3D software we used and Unity are not 1:1. Once imported, the art assets were attached to their relative Unity prefabs. Since creating art assets takes time, our programmers utilized simple programmer art that would function as a placeholder for the art assets. That way, once the art assets were finished, the programmers could simply replace the temporary art assets with the real ones.

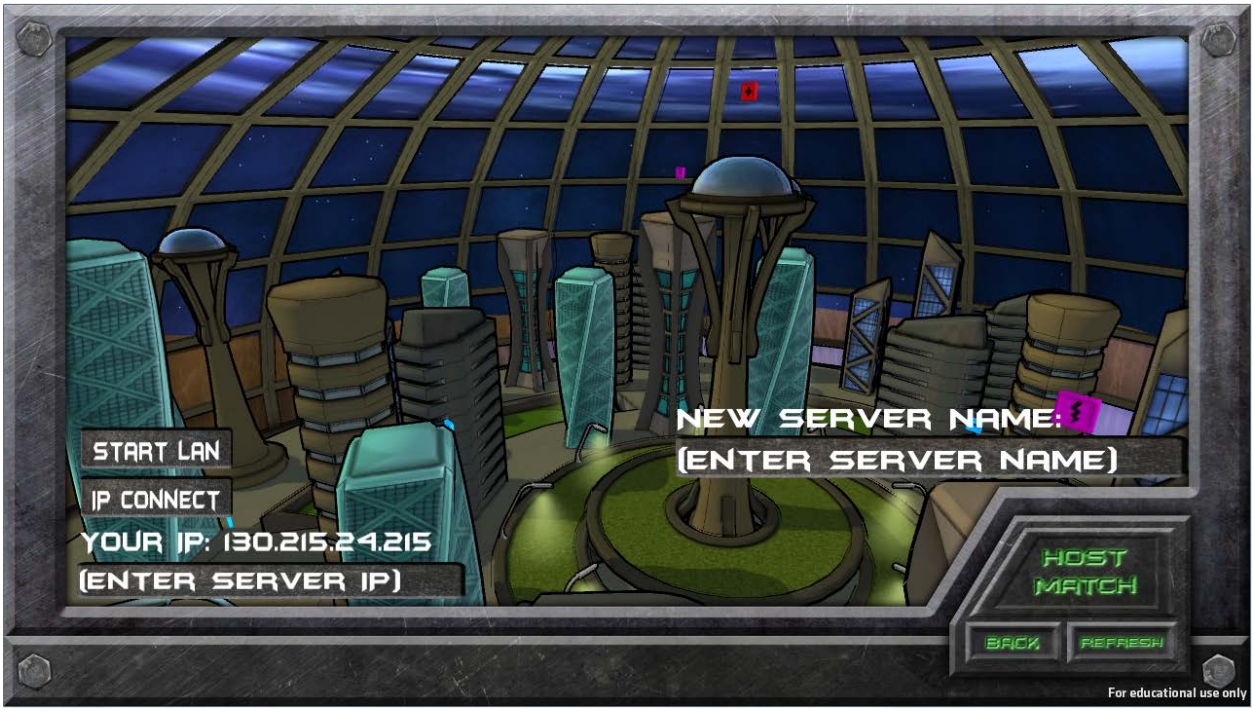
GUI and Menus

Main Menu

The Main Menu

The design of the main menu is a simple one following the theme of our game. The buttons are skewed to the left of the screen leaving the right side open to display an image of our game title. What the player sees when they start the game is the main menu, and from the main menu the player can navigate it to play the game, quit the game, or even view the credits.

Server Panel



The server-selection and creation menu

The server panel lets players browse for and create game servers. From the server panel, players can host a multiplayer game, start a Local Area Network (LAN) server, and connect to servers via browsing or IP address entry.

Character Selection and Spawn Panel

The character selection menu

After joining a server the player is presented with the character select panel. This panel also serves another function allowing players to choose when to spawn in the game. Players are given the option to select from two bikes, each with its own set of stats allowing players to see the differences between each bike not only visually but technically as well. With a bike selected, players can spawn into the game and begin playing.

Pause Panel

The pause menu

During gameplay, we wanted a way for the player to be able to leave the game or respawn when they please. We implemented a pause panel, which would display when the player presses the pause button (mapped to the “escape” key) in game. Since our game is multiplayer, pausing the game would disrupt play for other players, so we decided that our ‘pause’ panel would simply replace the players HUD with options to respawn or disconnect from the server, without halting game functions. Players can also view the IP address of the server they’re connected to, which can be used to invite friends to the game.

Player Heads-Up Display

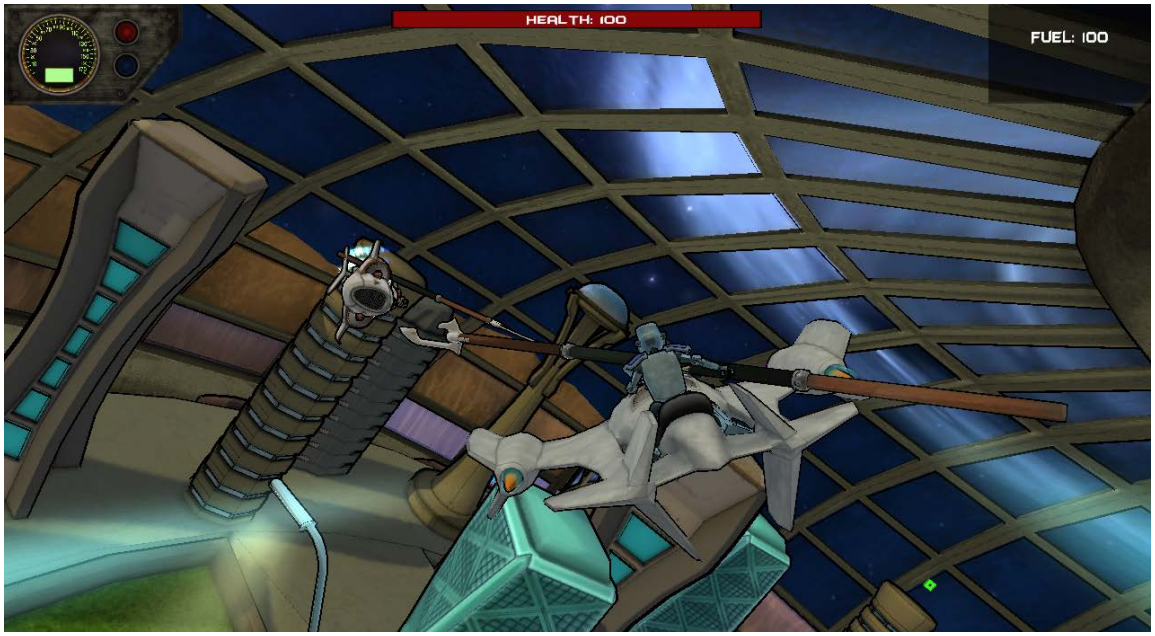
The Player HUD

For the player's heads-up display (HUD), we created a small panel for all of the essential pieces of information a player needs. The health is displayed both as an integer value and progress bar at the very top, and is clearly visible. A panel at the top-right of the screen displays the player's current pickup resources, such as the amount of fuel they have for the afterburner. As a cosmetic addition, we added a dummy panel at the top-left of the screen, which contains a red light that glows when the player has low health.

Playing Aeroknights: Deathmatch

While not in-game, players can choose between two bikes: the Aerobike or the Turbobike. The Aerobike is a highly maneuverable, albeit slow vehicle. The Turbobike is a high-speed but low-maneuverability vehicle. Once a bike is chosen, players can spawn into the game, where they will be placed on their chosen bike in a spawn point away from other players.

When players spawn into the game, they immediately have control of their character. Players can both both aim their weapon (the lance) and steer their ship with the mouse, and control the speed and roll of the ship with the keyboard. Every spawn point faces the center of the stage, encouraging players to fly to the middle where they are likely to find other players. Every bike has a bright-blue trail, making it easy to spot other players.



The player "jousts" his opponent in a head-on attack

When players spot each other, they can either immediately charge in and attack or duck away and find pickups to given them an advantage. Often, when both players choose the former, a sort of jousting match ensues where both player fly straight for each other, aiming to get the hit.

Top: The attacking player chases his opponent,

Bottom: the evading player sees his opponent through the rear-view camera

If a player spots an opponent before the opponent spots them, that player can tail the opponent and score a hit from behind. To increase spatial awareness, players can toggle a rear-view camera to see if anyone is behind them. Players being chased have two general strategies to shake their aggressor: They can attempt to fly evasively, pulling aerobatic maneuvers and weaving between buildings, and they can collect pickups, like afterburner fuel or mines, to deter their opponent. Mines are a useful way of aggressively deterring opponents, as they will directly deal damage. Afterburner fuel, on the other hand, provides the player with a speed boost which can be used to outrun their aggressor.

A player shoots at his opponent with the energy burst

Chasing players, meanwhile, can catch up to their opponents by outmaneuvering them and utilizing the energy-burst pickup. The energy burst is a projectile that slows down every player it hits, but does not damage them. Hitting the evading player with the energy burst can make it much easier for the chasing player to score a hit.

Every time a player is hit, they lose health and a “broken glass” screen effect is played. If their health drops below 0, they are destroyed and must spawn back into the game.

A player is struck and takes damage

A player is killed, leaving behind an explosion as his opponent zips past

More than two players can be playing at a time, so if a third player enters the game, they can intervene on a fight and score an opportunistic hit. Players must thus consider opportunistic opponents when singling out a target. If an opponent interferes with a player's attack or defense, the player would have to change their strategy to avoid being taken advantage of.

Testing

After the core mechanics were created, we conducted a play-test session. Play-test subjects were chosen randomly via mass-email and were asked to play our game for 30 mins. During play, we observed several things: the subjects' ability to control the bike, attack other players, and avoid getting hit by players. The three metrics were measured subjectively. We also noted subjects' comments while playing.

Our game's mouse steering and keyboard throttle seemed to be intuitive enough for the subjects to control the bike without confusion. Subjects were also able to consistently avoid obstacles. When it came to combating enemies, however, the subjects had a difficult time landing a hit on their opponent. During combat, the subjects' most common strategy was to simply fly straight for their opponent. This would often lead to two scenarios; either the subjects would fly straight at each other in a sort of joust, or one would flee while the other one chased. The former scenario would often result in both subjects hitting each other simultaneously, while the latter scenario would result in a long chase where neither subject made any significant progress.

When asked about their experience, the subjects expressed frustration in how difficult it was to hit their opponents. One subject suggested that the difficulty came from the maneuverability the bikes had at high speeds. One of the suggestions was to lower the turning radius of the bike as the throttle got higher in order to reduce the opponent's ability to evade. The subjects also suggested increasing the size of the target area on players to make them easier to hit, adding a rear-view camera to know if they are being chased, and adding some sort of projectile to use while chasing.

We followed up on the first suggestion by linearly reducing the bike's pitch and yaw controls as the throttle increased. At top speed, the bike would still be able to turn, but at

reduced magnitude. We additionally made the bike's pitch magnitude greater than the yaw magnitude. This would mean that, in order to get the tightest turn radius, players would have to roll so that the direction they're turning is relatively up. This would make it easier to predict what direction evading players would turn towards while trying to shake their opponent.

Another suggestion we followed up on was adding a rear-view camera. In order to let players know if they are being chased, we decided to completely follow the subject's suggestion. We added a rear facing camera to the player prefabs and wrote a script that would toggle this camera when the 'z' or 'c' keys are held down.

The final suggestion, however, seemed to go against the original intent of Aeroknights Deathmatch. Our goal was to create a flying game where melee weapons were the primary form of attack. We decided, however, that so long as the projectile was significantly less effective than the melee weapon, the core elements of our game would be upheld. We thus added a projectile weapon as a pickup that would reduce a player's speed on contact, but not damage him2. Like the mines, this projectile would be limited by a number of shots, which could be replenished by running into the projectile pickup. The projectile would be aimed by the mouse input, similar to the lance.

Results

Final Product

The final build of *Aeroknights: Deathmatch* is an intense dogfight where players constantly fight other players in a science-fiction dome-city. In order to beat their opponent, players must utilize the full aerobatic capabilities of their vehicle, of which there are two options. The first vehicle, the Turbobike, is a high-speed but low-handling vehicle that excels at making quick hit-and-run attacks. The second vehicle, the Aerobike, is a much slower, yet much more maneuverable vehicle that is great for making quick dodges and out maneuvering opponents. The balance between these vehicles creates asymmetric gameplay, with both vehicles having distinct advantages and disadvantages.

Players have many offensive and defensive options. While attacking, players can utilize pickups like Energy Bursts and Afterburner Fuel to catch up to and mess with their opponent. While evading, players can utilize pickups like mines to deter chasers, and heath to replenish their health bar. Evading players can also utilize the terrain of the map to lead their aggressors through dangerous obstacles, as well as fly unpredictably. All of this creates a balanced game where it is just as easy to deal damage as it is to avoid it.

Aeroknights: Deathmatch can be played in multiplayer either over the internet, or a Local Area Network. The menu systems implemented allow for seamless transitions between the main menu, setting up a game, choosing a vehicle, and play, and quitting.

Overall, the use of melee combat while flying creates a very up-close-and-personal style of gameplay not often seen in other flying games.

Scrapped Features

After creating the initial weapon, the lance, we experimented with other weapon types. During our initial brainstorming we came up with the idea of a short-range grappling hook that could yank enemies off their bike. We decided early that the extra animation and physics required to make this idea work was out of scope, so we reduced the idea to a whip, which would have a longer range than the lance. Unlike the lance, which is constantly pointed outward and can deal damage at any time, the whip would need to be "swung" in order to be able to hit opponents. After developing a prototype of the whip, we found that using a weapon which was not constantly brandished was extremely difficult to time with our flying mechanics. When we matched one of us with the whip against one of us with the lance, the lance would win every time. We thus decided to scrap the whip.

Unrealized Ambitions

Over the course of the development of *Aeroknights: Deathmatch,* several elements we had considered including in the game were conceptualized, however they did not appear in the final game build.

The foremost of these was creating additional environments for the players to explore and fight within -- the dome-enclosed city is the only map currently available for play. This resulted primarily from the number of custom assets required for the city that would not necessarily be reusable outside of that specific environment. Creation of fresh assets for an entirely new game environment was not something feasible in the amount of time we had for our project.

Aside from introducing additional game environments, we originally planned for additional detail within our default city dome environment, such as sky tunnels. Due to time constraints the addition of these tunnels did not come to fruition. Additionally, adding in "roads"

for the fictional city was another concept that we had floated. Roads did not materialize in our game for the same reason as the sky tunnels.

Early on in our project our team had considered multiple characters for the game and allowing the player to choose between each in a selection menu. This would have necessitated the construction, texturing, rigging, and animating of each of these characters. Instead, our team decided on differentiating the characters through color scheme - a red robot is associated with the Aerobike and a blue-robot is mounted atop the Turbobike.

Additional weaponry had been in the works earlier on in development of *Aeroknights: Deathmatch.* An example was the conceptual grappling hook launcher. Initially, we had considered an arm-mounted grappling hook launcher that would enable a player to pull an opponent off their vehicle or slow down the vehicle directly. This did not feature in the final version of *Aeroknights: Deathmatch* due to considerable technical challenges associated with implementing the weapon.

The scrapped grappling hook

Possible Extensions

Since *Aeroknights: Deathmatch* is, at its core, a versus-multiplayer game, there are many potential extensions that can be made in the future. Games like *Quake* have many game modes including deathmatch, where players constantly fight so see who can acquire the most kills; capture the flag, where players must team up to steal the opposing flag while protecting their own; and last-man-standing, which is similar to deathmatch, but once a player is killed, they do not respawn until the next round. Any one of these game modes would add variety to ours.

Deathmatch type games also benefit from having many different types of levels to choose to play on. The City Dome level included in Aeroknights is full of different types of buildings that can be navigated to avoid opponents. Other levels, such as a forest or cave, can be implemented to provide a unique experience that allows for replayability. For example, forest level will provide foliage and greenery that can hide opponents in plain sight. Whereas a cave level will provide a restricted play style with obstacles such as stalagmites and stalactites impeding movement.

Conclusion

The ultimate question we were trying to answer with *Aeroknights: Deathmatch* was “can melee combat work in a flying video game?” After six months of development, we have created a complete, standalone multiplayer game that is both fun and challenging. The flow of play is such that 1 on 1 battles last a decent amount of time, but do not drag on to the point of losing appeal. Given our efforts, we conclude melee combat in flying games is viable for a multiplayer setting, and can provide challenging content for single-player games.

Works Cited

Images

Air Conflicts: Secret Wars IGN Review. Digital image. N.p., n.d. Web. <http://xbox360media.ign.com/xbox360/image/article/121/1212525/shot01_1321391918.jpg>.

Pacific FIghters. Digital image. N.p., n.d. Web. <http://mediaserver.boonty.com/gamesimages/502_fr_sc1.jpg>.

Pegasus Knight. Digital image. N.p., n.d. Web. <http://vignette3.wikia.nocookie.net/fireemblem/images/6/62/FE9_Pegasus_Knight_%28Marcia%29.png/revision/latest/scale-to-width/640?cb=20121019124951>.

Pitch, Yaw, and Roll Diagram. Digital image. N.p., n.d. Web. <http://www.toymaker.info/Games/assets/images/yawpitchroll.jpg>.

Rogue Leader. Digital image. N.p., n.d. Web. <http://www.stealthybox.com/wp-content/uploads/2014/05/Star-Wars-Rogue-Leader-Death-Star_D5.jpg>.

Star Fox 64. Digital image. N.p., n.d. Web. <http://resource.mmgn.com/Gallery/full/Lylat-Wars-Star-Fox-64-DM.jpg>.

Star Wars: X-Wing. Digital image. N.p., n.d. Web. <http://static.gog.com/upload/images/2014/10/b262918263290b76bdaaa35a4511684abe962dd2.jpg>.

P-38. Digital image. N.p., n.d. Web. <http://upload.wikimedia.org/wikipedia/commons/b/bd/Lockheed_P-38_Lightning_USAF.JPG>.

Websites

"The Last Phoenix - The Plains Gameplay [Indie Game Prototype]." *YouTube*. YouTube, n.d. Web. 05 Mar. 2015. <https://www.youtube.com/watch?v=4e1Qj-e3rhc>.

"NGUI: Next-Gen UI Kit." *Tasharen Entertainment*. N.p., n.d. Web. 05 Mar. 2015. <http://www.tasharen.com/?page_id=140>.

Unity. *Unity*. N.p., n.d. Web. 01 Mar. 2015. <http://unity3d.com/>.

"Prefabs." *Unity*. N.p., n.d. Web. 05 Mar. 2015. <http://docs.unity3d.com/Manual/Prefabs.html>.

"What Are the Differences Between First-Person Shooter and Third-Person Shooter Games?" *EBay*. N.p., n.d. Web. 05 Mar. 2015. <http://www.ebay.com/gds/What-Are-the-Differences-Between-First-Person-Shooter-and-Third-Person-Shooter-Games-/10000000177589743/g.html>.

"Dieselpunk." Wikipedia. Wikimedia Foundation, n.d. Web. 06 Mar. 2015. <http://en.wikipedia.org/wiki/Dieselpunk>

Games and Films

Star Wars: X-Wing: LucasArts. LucasArts. 1993. Video Game

Freespace 2: Volition, Inc.. Interplay Entertainment. 1999. Video Game

Wing Commander: Origin Systems, Inc.. Origin Systems, Inc. 1990. Video Game

Star Wars Episode IV: A New Hope. Dir. Lucas, George. Twentieth Century Fox, 1977. Film.

Independence Day. Dir. Roland Emmerich. 20th Century Fox, 1996. Film.

World of Warcraft: Blizzard Entertainment. Blizzard Entertainment. 2004. Video Game

Fire Emblem: *Shadow Dragon and the Blade of Light:* Intelligent Systems. Nintendo. 1990. Video Game

The Last Phoenix: BearInMind games. BearInMind games. 2013. Video Game

Halo: Combat Evolved: Bungie. Microsoft. 2001. Video Game

Starfox 64: Nintendo EAD. Nintendo. 1997. Video Game