Creating a Competitive Multiplayer Open-Arena 2D Twin-Stick Shooter

Anthony Cloudy and Squirrel Eiserloh

***Abstract*— This thesis intends to demonstrate an all-around mastery of the lessons and skills developed through the Guildhall's software development track by building a competitive local multiplayer, open-arena, two-dimensional twin-stick shooter from the ground up. The artifact draws inspiration from *Kirby Air Ride*'s "City Trial" and attempts to create a more competitive and progression-focused game. *AllStar* demonstrates a holistic understanding of video game programming, including gameplay systems, engine coding, shaders and graphics programming, all while emphasizing polish.**

***Index Terms*— Game Development, Real-time Rendering, Software performance, Software quality**

I. Introduction

Mastery of programming for game development requires a broad knowledge base covering a multitude of difficult and varied skills. While much of the software development industry requires more niche roles with less cross-pollination of disciplines, gameplay and engine programmers must understand everything from input and real-time rendering to advanced data structures and networking. This thesis intends to demonstrate a mastery of the lessons and skills developed through the Guildhall's software development track by building a competitive multiplayer, open-arena, two-dimensional twin-stick shooter from the ground up. The project aims to demonstrate an all-around mastery by creating a well-polished game, *AllStar*, in a custom C++ engine.

The artifact applies optimization techniques to create a performant gameplay experience for up to four players. The game pushes its engine's codebase to its limits, and demonstrates the extents of the author's engine built from his Guildhall experience. The game features split-screen local multiplayer that pits players against one another in an arms race to build the most powerful ship within a set time. Players explore an open arena, destroying cargo crates and enemies to earn upgrades to their ships. Upgrades affect the base stats of a player's ship, enabling her to go faster, tank more damage, or shoot more powerfully. The player can hunt others in this game mode to steal some of her opponent's resources, and all player's stats and equipment are locked in once time is up. The player then must use her powered-up machine in three randomly-chosen contests: including but not limited to a battle royale, a race, or a coin-grabbing challenge. Because the contests are chosen randomly, the player has no idea what kinds of minigames she's going to compete in, which adds to the frantic and fast-paced nature of the game.

The artifact was built using a public GitHub repository to log all code commit messages, which helped discern what was done when and kept progress transparent. A development diary was kept not only to capture what challenges arose during development, but to also document decisions and problems resolved during the artifact's creation. A concentric development approach divided the game's feature set into tiers, defining clear stages for the project and creating milestones.

II. Research Review

Because of the competitive nature of this artifact, this literature review focuses primarily on finding resources on competitive game design and creating multiplayer experiences. This also includes research into specific challenges the artifact faced, including split-screen game design and shoot-em-up (*shmup*) design. Games that demonstrate a strong competitive/multiplayer design are of equal importance to the research, and provide proven examples of what works and what does not. Other games are included in the research review for their specific shmup qualities or control styles that provide reference for the artifact's design. The researcher consulted professors Squirrel Eiserloh and Christopher Forseth from The Guildhall at Southern Methodist University to find games with mechanics, playstyle, and gameplay similar to the proposed mastery project. The researcher then studied the suggested games, including *Galak-Z* and *Realm of the Mad God*. The researcher also utilized his personal games library to find more similar games, such as *Kirby Air Ride*. Finally, the researcher utilized Bing and the SMU Central Libraries to research "Competitive Games", "Competitive Game Design", and "Splitscreen Game Design" to find articles and research various aspects of gameplay the artifact will utilize. The researcher

Anthony Cloudy is with Southern Methodist University Guildhall, 5232 Tennyson Parkway, Building 2, Plano, Texas 75024 USA (e-mail: acloudy@smu.edu). He graduated from Southern Methodist University in 2015 (B.A. Computer Science), and has previously done contract work with Fractal Fox as a professional game developer.

Squirrel Eiserloh is a game programming faculty Lecturer at SMU Guildhall, Southern Methodist University's game development graduate program. Since he graduated from Taylor University in 1996 (B.A. Physics) he has been working as a professional game developer in the Dallas area, contributing to over a dozen commercial game titles. He co-chairs the Dallas chapter of the IGDA, and coordinates the Math for Game Programmers sessions at the annual Game Developers Conference in San Francisco. (e-mail: beiserloh@mail.smu.edu)

limited search results and games only to those localized in English.

A. Literature Review

In "Rock Paper Scissors - A Method for Competitive Game Play Design", author Victor Chelaru discusses the nature of Rock Paper Scissors (RPS) design in games, in which certain attacks have an absolute advantage or tie with others (just like the game the design's namesake shares). The article goes in-depth on the metagame of "Pure RPS", where the attacks have no lead up or predictability (grounded units vs flying units in an RTS), "RPS and Signals", where attacks do have *readability* (such as the wind-up animation of a punch), and "RPS with separate Attacks and Signals", where attacks have signals, but experienced players can cancel or feint signals. The article reveals the emerging dominant strategies for RPS games, and discusses ways to keep the game from incentivizing undesired player behaviors. Because some dominant strategies include "be random and fast" and "don't initiate any attacks" for the more basic RPS designs, ignoring the insight this article has could destroy the metagame, and thus was considered for this artifact. The dominant strategy that evolves from the most advanced RPS design is to adapt to one's opponent's patterns, which encourages a healthy, competitive game that prioritizes player skill and reading one's opponent without promoting stale tactics [1].

The article "Shared-Multi-Split Screen Design" [sic] by Richard Terrell assesses and compares the distinctive design considerations and limitations provided by various types of multiplayer screen layouts. The article exposes some of the tradeoffs and design challenges that split-screen games face. Split-screen gameplay can force a reduction in graphical quality, as the game must render two to four separate views every frame. The reduced screen space also can cause problems for players, as this space conveys important spatial information. Other design hurdles mentioned in the article include the introduction of screen-peeking, a need for increased monitor size to prevent feeling constrained, and increased team communication if players want to cooperate. The article fails to mention any positive aspects of split-screen as opposed to multiple screen, which include cheaper setups, greater flexibility when playing with other people, zero network latency, and the potential for more positive experiences that come from playing with others in person [2].

The postmortem for *Good Robot* provides valuable insight into some of the unique design challenges shmups face. The developer, Shamus Young, started the game as a solo project, but eventually transitioned to work with another studio once he realized that the game's design had issues. The postmortem outlines how he managed to resolve the game's flaws by working with the other team's ideas, which included establishing a dynamic gameplay rhythm, with valleys and peaks of activity, and adding consequence to player death. Many of Young's concerns are pitfalls this thesis had to avoid during development, especially in regards to game design and mechanics not panning out or a lack of proper pacing. Failing to give players the sense of enjoyable tension, or failing to create meaningful and interesting player interactions can endanger similar projects [3].

A paper titled "Group Report: Progression Systems" from Project Horseshoe 2014 deconstructs the nature of progression systems. The report broke progression systems down into a series of building blocks that make up system fundamentals, as well as tactics to strengthen player motivation towards interacting with the systems. Of the system building blocks described, the most relevant to this thesis include *progression loops*, which spiral upward as players gain power in order to accomplish new feats which grant them new powers. The power up system, in which players continually make incremental improvements on their ships, matches a power loop, where playing the game improves the player's avatar's power, which improves their "virtual skill" for the round. The paper also links player motivations, such as superiority and control, to rewards like competition and power, via *progression atoms*. Progression atoms are in-game components that serve as the conversion from the player's motivations into rewards. By giving a player who wants better control of their character a set of character stats, they can give the player the reward of power through those stats [4].

B. Field Review

Figure 1: *Kirby Air Ride's* City Trial mode features power-ups scattered throughout the level that alter the characteristics of players' machines *[5]*.

Kirby Air Ride is a multiplayer 3D racing game created for the Nintendo GameCube, which puts players head to head while piloting a variety of quirky "Air Ride Machines". The game features an alternate game mode called "City Trial", in which players are put in an open map and given free roam for 5 minutes. Players begin on a basic, neutral Air Ride, and are tasked with finding a better machine and collecting power-ups to customize their machine within a time limit. At the end of the round, all players compete in a random minigame that tests the player's skill and powered-up ride, with the winner of the minigame winning the whole game. Although Terrell's paper describes many of the design limitations of split-screen, *Kirby Air Ride* manages to utilize split-screen successfully to create an enjoyable experience despite these limitations, and many of the performance tradeoffs are either hidden by the game's design or minor incidents (such as a few occasional framerate hiccups). The artifact for the thesis draws heavily upon City

Trial's gameplay for inspiration, and aims to push the boundaries of this original idea and take it to a new level. This thesis attempts to utilize *Kirby Air Ride*'s unique gameplay style that provides randomness without arbitrary outcomes, while addressing the game's minimal player interactions and unwieldy combat [5].

Figure 2: Players battling monsters in *Realm of the Mad God* have to pay extremely close attention to their surroundings, as bullets come in various speeds and patterns that can end players' lives instantly *[6]*.

A fantasy bullet-hell with fast leveling and *permadeath* (the game deletes a player's character when they die), *Realm of the Mad God* is an unconventional massively multiplayer online game (MMO). Players are thrust onto an open world in which they travel to defeat enemies, gain experience, and loot corpses until all major bosses on the map have been vanquished. Once the players have defeated the bosses, the whole server is thrust into a battle with the game's final boss. The game is a twin-stick shooter, in which players avoid bullets while desperately trying to land shots on the hordes of enemies. *Realm of the Mad God's* map and player versus environment (PvE) combat line up a significant amount with the design of the thesis artifact. Whereas players are incentivized to defeat enemies through the chance of rare equipment upgrades in the MMO, the artifact aims to use the power-up system to incrementally boost the player's stats. The artifact gives out a multitude of small power-ups with few and far-between equipment pickups, instead of a constant stream of class-specific equipment you may or may not be able to use. *Realm of the Mad God*'s pickup and equipment system is also important to the thesis, as players are inundated with a steady supply of weapons, armor and potions at a rate that matches the quick-paced nature of the game [6].

Figure 3: A player avoiding Sinistar while trying to create sinibombs. The game's open arena and obstacles match the thesis' design *[7]*.

Sinistar is a top-down, multi-directional shooter where the player is locked in an arms race against "Sinistar", the game's villain. While enemy workers attempt to reconstruct Sinistar, the player attempts to survive gunfire and mine planetoids to create "Sinibombs", the only weapon that can defeat Sinistar. Once Sinistar is created, the player needs either to destroy him or run away, as getting caught by Sinistar results in instant death. *Sinistar* provides an example of a PvE RPS balance that shifts over time, as it requires players to juggle mining, direct attacks, and evasive maneuvers to win against Sinistar [1]. The act of mining leaves the player open to attacks from warriors, but without sinibombs, players can only evade Sinistar, as his attack trumps the player's standard laser. The player's options change in value before and after Sinistar is activated, creating gameplay dynamics that change over the course of the play session. *Sinistar* has very similar theming, handling, enemies and obstacles to those in *AllStar*. The act of shooting level obstacles to acquire resources, the way the player navigates through the level, and the tension felt during combat in Sinistar match many of the thesis' core mechanics. However, while *Sinistar* generates tension via PvE, the artifact generates this tension mostly via PvP, as the arms race is between players, not an almighty boss [7].

Figure 4: *Galak-Z*'s unique handling and polish set it apart from other titles in the genre, creating the feel of actually driving a spaceship *[8]*.

Galak-Z is an, 80's sci-fi anime styled roguelike shmup that casts players as a lone pilot fighting against enemies in cavernous planetary dungeons. The gameplay combines roguelike gameplay with shmup controls to create a unique experience, as the player pilots a physics-based ship through various "dungeon rooms". The game also values stealth, as the player's rockets make noise that alert enemies to the player's presence. The game's unique aesthetic and polish are high quality, and while mostly out of scope for the constraints of the thesis, served as a great reference to aspire and work towards. *Galak-Z*'s ship controls are also intuitive, and the artifact aimed toward a comfortable medium between the game's physics-based motion and *Realm of the Mad God*'s point-and-move control scheme [8].

The majority of the games listed have some sort of RPS gameplay, as outlined by Chelaru's paper [1]. *Kirby Air Ride* features jousting-based combat that utilizes RPS with separate attacks and signals, as players must approach one another to attack, and can easily feint an approach to sway their opponent's behavior [5]. *Kirby Air Ride*'s constant stream of power ups grants the players more control, but the game fails to provide progression systems for other common competitive player motivations [4]. The flow and rhythm concerns that arose during the development of *Good Robot* are an obstacle some of these games overcame as well [3]. Although extremely hectic, *Realm of the Mad God* manages to establish this rhythm through the spacing of enemies in dungeons, and by giving the players the ability to break out of tight situations via instant teleport to a hub world [6]. Because players are able to lure and stack multiple enemies to create hordes that would obliterate the game's flow via incredibly intense moments, giving the player the option to take a break at any point prevents the game from becoming overwhelming [6]. *Galak-Z* comes from the other end of the spectrum, where the majority of gameplay isn't hectic, but tension and flow is generated through stealth and using level obstacles to alleviate pressure.

C. Summary

This artifact aims to create interesting competitive multiplayer gameplay while attempting to avoid the various pitfalls and issues discovered through research. By utilizing RPS with separate Attacks and Signals as a foundation for designing player options and interactions, the artifact can avoid stale or boring dominant strategies [1]. Without the separation of attack and signal, the best course of action becomes never initiating attacks, which detriments the game [1]. Although the thesis intends to be competitive, the players should not always be at each other's throats, as mentioned in the postmortem for *Good Robot* [3]. This thesis attempts to establish a good gameplay rhythm by balancing player interaction with the map's scale, allowing players the choice to fight and the space to run off and recover, without making the map too large for players to find one another. While *Kirby Air Ride* creates an interesting play space and encourages moments of interaction through gameplay events, the game fails to incentivize combat enough. Players must be extremely close to one another to consistently battle, and with the scale of the map and handling of the machines, the game fails to deliver an incredible PvP experience [5]. This project attempts to combine *Realm of the Mad God* and *Galak-Z*'s differing control styles to create the best combat experience for the artifact. As mentioned in Terrell's article, split-screen has a host of downsides and technical limitations that the artifact works to overcome [2]. Considering that optimization is a part of the mastery the thesis intends to demonstrate, the project attempts to ensure that the game runs well even with 4 players on screen. *AllStar* attempts to combine the best parts of *Kirby Air Ride* and *Sinistar* with a hybrid control scheme based off of *Realm of the Mad God* and *Galak-Z*.

III. Methodology

The artifact is designed to demonstrate the author's mastery of the teachings and concepts taught in SMU Guildhall's programming track. The project demonstrates gameplay, graphics, and engine programming, as well as the ability to create procedurally generated content, optimize, and polish a game through code.

A. The Game

AllStar is a competitive, open-arena, two-dimensional twin-stick shooter that runs in a custom C++/OpenGL game engine. The game is for two to four players and is controlled using one to four Xbox/XInput controllers. The game has support for keyboard/mouse and single player matches for the sole purpose of debugging (which would be removed if this were a commercial build). Each player flies their ship using the left joystick, while aiming and firing their weapon with the right. The left trigger activates any active abilities the player has, while the right trigger teleports the player.

Game Flow

A game of *AllStar* lasts a total of 10-15 minutes. Players start on the Player Join screen, where each can pick his or her ship color and "ready up" for the game. Once all players are ready, gameplay goes through two phases: Assembly and Challenge.

In the Assembly phase, each player flies around an open arena and scavenges for upgrades to his or her ship. In the Challenge phase, the player plays against her opponents through a trio of minigames using her upgraded ship to fight for victory. After finishing the final minigame and viewing the game's overall winner, players are returned to the title screen where they are given the option to play again and try out new strategies and combinations of upgrades.

Figure 5: A game of AllStar, featuring the two main phases of gameplay, Assembly and Challenge.

Assembly

At the start of the Assembly phase, each player starts with a default ship with no stat modifications, and is given five minutes to assemble her ship. The player roams an open arena, trying to find as many *power-ups* and *equipment* as she can to build a ship that suits her style. Power-ups are pickups that modify a player's stats, while equipment are pickups that change the player's abilities, weapon, and base stats. The specific combination of these pickups compose a player's *build*: the balance of skills based on boosts from equipment and power-ups that describes how the player's ship is most likely to fare in various minigames. For instance, a defensive build would have high defensive skills, but comparatively fewer speed and attack skills, meaning the ship would have the advantage in a battle, but have the disadvantage in a race.

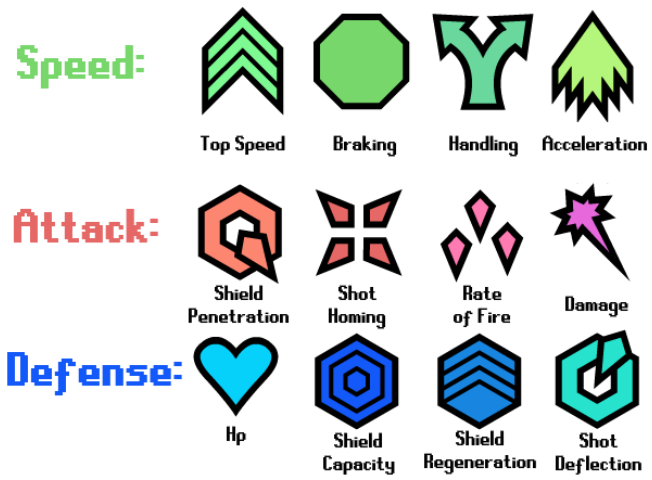

Figure 6: The twelve power-ups, in their respective power families.

Power-Ups

Picking up a power-up increments one of the player's twelve passive skills, such as top speed or shield regeneration rate. Power-ups are grouped into three families: speed, attack, and defense. Each family has four power-ups that affect the player's stats in a related manner. The speed family includes: *top speed*, which increases a player's maximum velocity; *braking*, which decreases the amount of time for a player to come to a complete stop; *handling*, which reduces the time it takes for the player to change her direction of motion; and *acceleration*, which improves how quickly a player reaches maximum velocity. The attack family includes: *shield penetration*, which increases a player's damage bonus when attacking shields; *shot homing*, which increases degree to which shots home in on enemies; *rate of fire*, which decreases the cooldown time between shots; and finally *damage*, which increases the amount of damage each projectile does to other entities. The defense family includes: *Hp*, which increases a player's maximum health; *Shield Capacity*, which increases a player's maximum shield health; *Shield Regeneration*, which increases the rate of regeneration of shield health outside of combat; and *Shot Deflection*, which increases the degree to which shots are pushed away from the player as they approach. Each player is trying to get as many of these power-ups as she possibly can, as each power-up improves her stats for the remainder of the game. A player can find power-ups by breaking crates, destroying asteroids, or defeating non-player enemies.

Equipment

Players are also on the lookout for *equipment*, which are pickups that provide a player with new abilities, weapons, and temporary stat bonuses cumulative with those gained by power-ups. Each player has four swappable equipment slots, one for each type of equipment: the *active*, a special ability that a player activates using the left trigger; *weapon*, the projectiles the player fires when shooting; *passive*, a gameplay-modifying bonus or always-on ability; and *chassis*, which is the player's ship body. A player can find equipment by destroying any of the crates scattered around the map, which has a chance of containing any of the above equipment with the power-ups dropped.

Chassis

The chassis is the foundation of a player's build, as it has the most impactful skill bonuses and drawbacks of all equipment. For example, the speed chassis dramatically improves the player's top speed and acceleration, but reduces handling and damage significantly, allowing the player to move quickly in straight paths, but turn slowly in wide arcs.

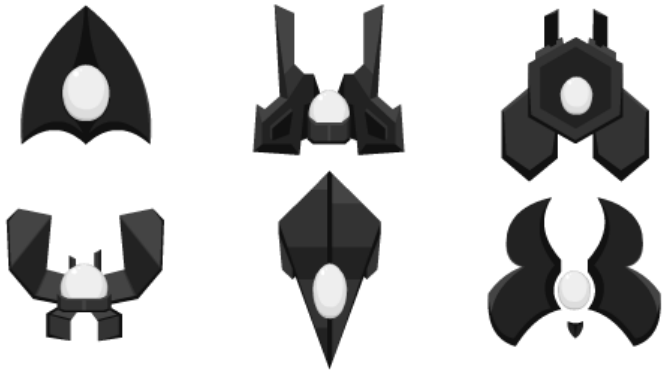

Figure 7: Concept art for the different chassis in the game.

Weapons

Each weapon shoots different types of projectiles, and has a large impact on how the player approaches enemies and other

players. Certain weapons like the missile launcher encourage area control, while others like the wave gun encourage precision and proper spacing, resulting in different optimal strategies for each build.

Figure 8: The spreadshot weapon (right) encourages players to get in as close as possible to blast enemies with as many shots as possible. The wave gun (left) outranges the spreadshot, but encourages the player to keep enemies at the focal point for optimal damage.

Passives

Passives are equipment that provide a gameplay change passively, such as a player being able to cloak whenever her ship isn't moving. Picking up the Spray and Pray passive encourages more area coverage with projectiles through increased rate of fire at the cost of reduced damage per projectile.

Actives

Actives are activatable equipment that give players new abilities which grant a temporary edge over other players. These abilities range from temporary power-up boosts – like Quickshot's large burst in rate of fire for 5 seconds – to more custom actions, such as Boost's ability to dash and deal damage on contact with other entities.

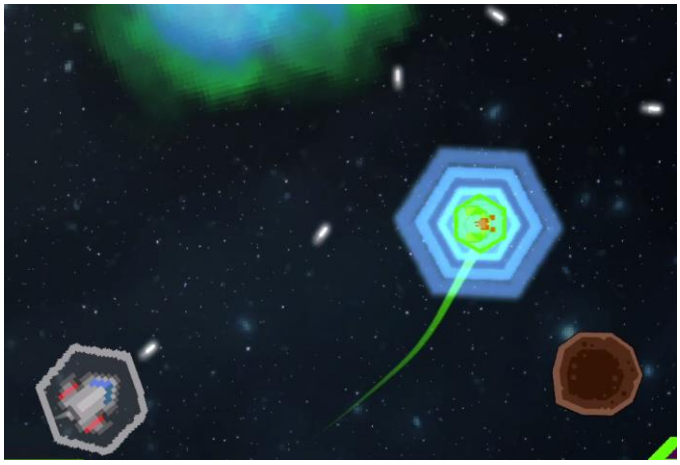

Figure 9: A player using her active ability to drastically increase shot deflection.

The Assembly arena is filled with major and minor *encounters* – procedurally-generated landmarks, enemies, and features that populate the world. Players can seek out encounters that let them customize their builds towards different goals, such as enemies or crates that drop speed power ups. A player can also stumble upon environment landmarks, such as detection-suppressing nebulae and teleporting wormholes that create interesting strategical advantages and disadvantages. If a player dies during the Assembly phase, her chassis is destroyed. The dead player also drops a percentage of her power-ups, and potentially one of her non-chassis equipment pieces, which can be picked up by other players. The defeated player is able to respawn immediately with the starter chassis and resume the arms race with the power-ups and equipment she has remaining.

Figure 10: A player taking advantage of a nebula to sneak up on an enemy ship.

After the Assembly phase ends, players view a summary of their power-up stats on a results screen, displaying what each of them gathered. The players are then locked in to those power-ups and equipment for the remainder of the game. After this menu, the Challenge phase begins, and each player must compete with her newly assembled ship for a chance at victory.

Figure 11: One of the minigame splash screens.

The Challenge phase of gameplay consists of three minigames. This gameplay phase takes a player's ship build and challenges her skills on a variety of different factors. Since each minigame is randomly selected, a three-minigame format helps to prevent a player from losing the entire game due to minigame selection (e.g. a slow, defensive build being subjected to a race). Each minigame challenges the player in a variety of ways, from battling to drag racing, from grabbing coins to a fight to the death around a growing black hole.

Each minigame lasts no more than two minutes, and players earn points based on their ranking in that game. The 1st place player wins seven points, the 2nd earns four, 3rd gets two, and 4th is awarded a single point. If any players are tied for a place, they each receive the same points (two 1st place winners would both

get seven points, while the next player would get 3rd place's two points). The player with the most points at the end of the three minigames wins. However, if at the end of the three minigames two or more players are tied, a sudden death minigame is played. This consists of a small, empty arena where the tied combatants fight to determine the sole victor. If the sudden death ends in another tie, the game runs sudden death minigames until a single 1st place winner has been selected.

Figure 12: The sudden death minigame mode. If there's a tie for points at the end of the game, the tied players are thrown into this minigame for one final battle.

B. *Program Structure & Techniques*

AllStar utilizes inheritance to quickly and easily add new entity interactions and gameplay functionality. *TheGame* class manages the game's state and carries players across the different game modes. Each *GameMode* handles the game logic (for Assembly or any one of the minigame modes) and updates the entities in the game world. TheGame handles the transfer between GameModes and results screens, and defers to each GameMode to handle gameplay and player updating logic. Everything spawned in the game world is an *Entity*, and uses inheritance to share functionality.

GameModes

Each *GameMode* owns a world and is responsible for running gameplay in the game. Subclasses of GameMode handle creation and initialization of the world and entities, and keeps them all within the map's bounds while updating the camera and other gameplay elements.

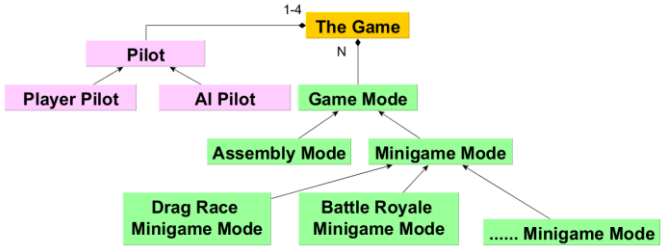

Figure 13: A UML Diagram that displays the core gameplay architecture of the program.

Each GameMode handles the procedural generation of its game world by populating the world with *Entity*s (props, enemies, and players). Each GameMode handles and updates all the entities in the game every frame.

SpriteGameRenderer

Each entity has a S*prite*, which is automatically registered with the *SpriteGameRenderer*, an engine subsystem that handles the bulk of the game's rendering. At program startup, TheGame grabs all the game's required textures and loads them in as *SpriteResource*s, each of which contains the base information required to render a specific sprite. Whenever a gameplay element requires a renderable component, it creates a new *Sprite* object. Each *Sprite* references a SpriteResource object registered in the engine's *ResourceDatabase*, which owns all preregistered assets. Creating a *Sprite* object automatically registers it with the SpriteGameRenderer on a rendering layer, after which it begins rendering automatically. The destruction of the sprite object also removes it from the rendering layer automatically. For a more detailed explanation, please refer to the appendix.

Entities

The Entity base class contains the core functionality for objects in the game world, which includes moving, taking and receiving damage, calculating and resolving collisions, and more. *Ship*s, *Projectile*s, and *Pickup*s each directly subclass from Entity, each expanding on the functionality in a unique way. Bullets fired by ships are Projectile objects, which override collision detection functions to disappear after dealing damage. Pickups are the physical representation of *Item*s in the world. Each pickup has its own item payload, which is transferred to a player upon colliding with that pickup. Items on their own can't be rendered in the world, but once wrapped by a Pickup, they gain a physical presence (a transform, and a sprite).

A *Ship* is an entity that has a *Pilot* and can fire projectiles. A *Pilot* is a class that contains the virtual input for a specific ship, and moves the ship around. *Ship* subclasses include *PlayerShip* and any individual Enemy ship classes, such as *Grunt*. Ships differ from entities in that they have more complex movement options, which are read from their *Pilot*. TheGame initializes the *PlayerPilot*s during the ship selection screen, based off which controller (or keyboard, for debugging) the player is using. TheGame creates an *InputMap* based on the player's input device and binds physical inputs to virtual inputs. Whenever a ship wants to update its position, it polls the pilot's input map to find the direction in which the ship is moving, and any other inputs needed to complete the update.

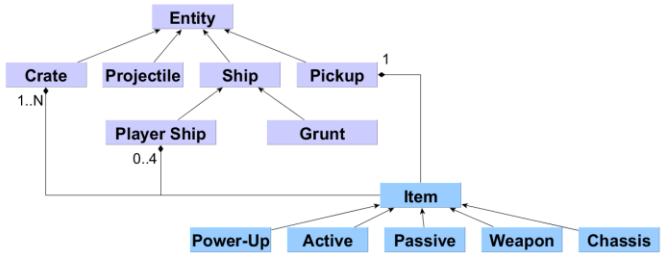

Figure 14: A UML Diagram that shows the relationship for entities and items.

Procedurally generating the maps was chosen over designing individual levels due to the programming-focused nature of the

thesis, and time constraints. The process starts by adding anywhere from 50 to 100 asteroids to the map by randomly picking spots inside the arena.

After filling the map with asteroids, the game determines a set number of *encounters*, or map features, to be spawned in the game. Each game mode picks how many encounters are in the map, and can control the amount and types of encounters it spawns. Types of encounters include: *nebula*, which cover up part of the gameplay area in colorful clouds, obscuring players, enemies, and items behind them; *bossteroid*s – huge asteroids that act as obstacles and a source of many smaller asteroids; *black hole*s, which suck entities into their center and destroy them; and *wormholes*, which suck in entities towards their centers, but spit them out harmlessly through another linked wormhole on the map.

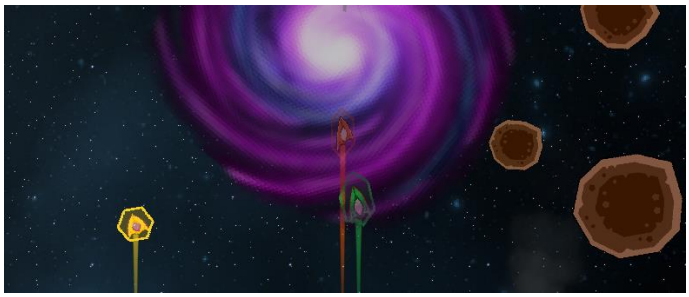

Figure 15: Players getting sucked into a wormhole. Once they reach the center, they'll be shot out of the other corresponding wormhole on the other end, which could be anywhere else in the map.

The game splits encounters into two groups, *minor* and *major* encounters, based on the physical size and gameplay impact of the encounter. For example, as a nebula is less gameplay-impacting and more passive, it is a minor encounter and spawned more frequently. Conversely, wormholes and black holes take up much more play space and actively impact how players play the game on a much larger scale, and are thus large encounters and limited in the number of spawns they have in the world.

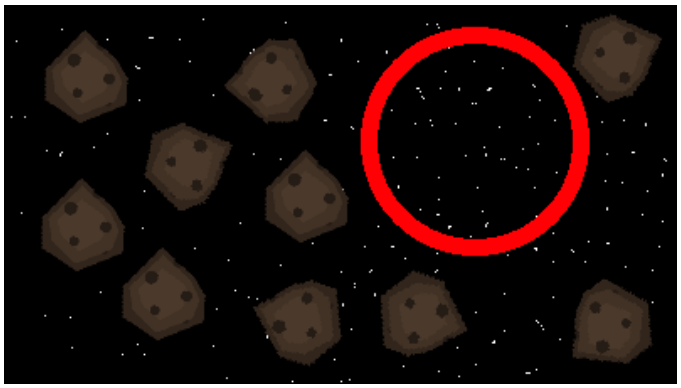

Figure 16: The GameMode clears out any entities within the radius of the encounter, which removes any asteroids that would be colliding with this new encounter.

After selecting an encounter, the game generates a random radius and attempts to spawn the encounter into the game. The game spawns the major encounters first, then moves on to the minor ones.

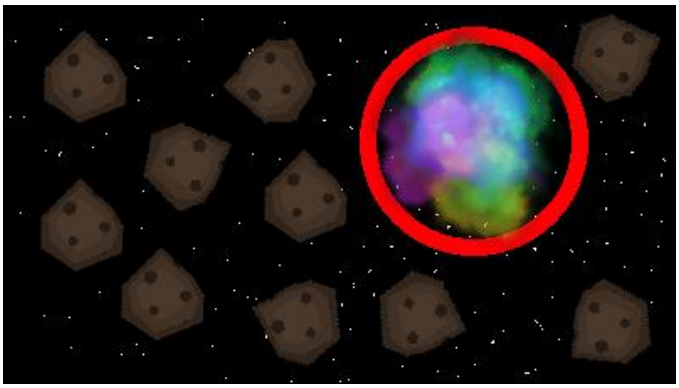

Figure 17: The encounter is spawned in.

Once the GameMode has selected a valid location for the encounter that doesn't collide with any other encounters, the GameMode deletes any entities within the proposed encounter's radius. The process checks for collisions with any of the entities on the game map, and removes anything that could potentially interfere with the encounter. Finally, once the area is cleared, the GameMode spawns in the encounter. Each encounter object is coded using relative coordinates, which allows the entities within an encounter to be placed in a regular pattern based on the scale of the radius the cleared-out space.

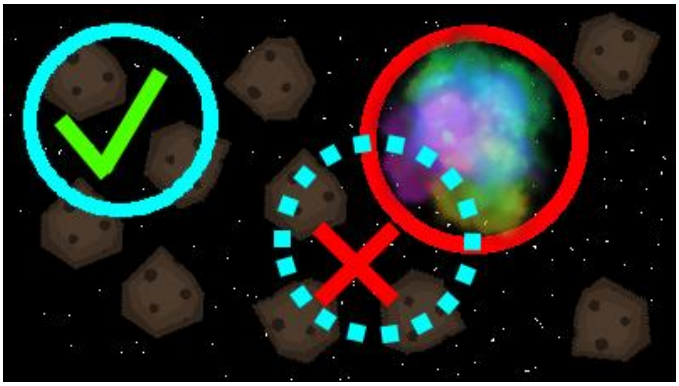

Figure 18: A new encounter attempting to spawn in collides with a previous and fails. A second attempt is made that collides with no others, and succeeds.

Subsequent encounters are spawned in checking against all the previous encounters' boundaries. This step is to ensure that no entities of another encounter are removed when clearing space for a new encounter. In the figure above, an encounter's random location is too close to our previous encounter, forcing the encounter to pick another location.

C. *Development Process*

Game balance was a consistent struggle throughout the project. A spreadsheet was created to iterate on and test different power-up values, in order to simplify game balance and better expose the function and dependency of the game's power ups. *AllStar's* power-up stats range from 1 to 36 internally and from -5 to 30 externally (from the player's perspective). Initially, stats grew linearly, which proved insufficient for balancing the project. Stats have an option of multiple curves to create a better growth trajectory.

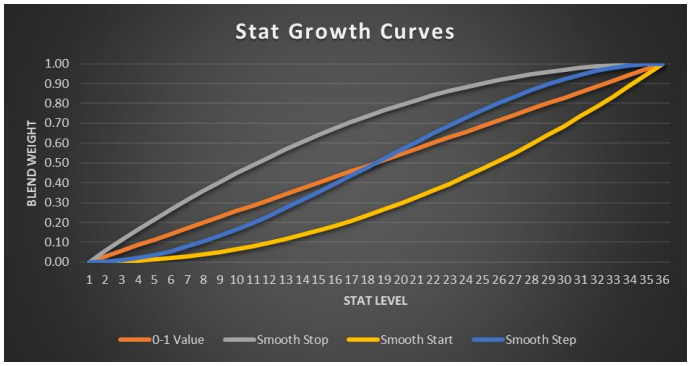

Figure 19: The potential stat growth curves for a stat.

The graph above shows the potential stat growth curves that each of the skills could follow. While the skills started off growing linearly, the level discrepancy grew quickly and caused huge power gaps between players with 0 and 5-10 power-ups. Thus, geometric growth grew to be crucial for helping to prevent early-game snowballing. Most of the skills ended up following the Smooth Stop (ease out) trajectory, but a few implemented Smooth Start (ease in) to prevent the major effects from revealing themselves too early on. Fresh characters start with stats at level 6, and after collecting the maximum number of power-ups for a stat (20 power-ups) players reach level 26. Equipment bonuses can push players up an additional 10 levels over the maximum stat, with level 36 as the absolute max level.

Stat Level	Top Speed	Stat Level	Top Speed
1	2.00	19	8.78
2	2.03	20	9.33
3	2.12	21	9.88
4	2.27	22	10.42
5	2.47	23	10.95
6	**2.72**	24	11.46
7	3.02	25	11.95
8	3.35	**26**	**12.42**
9	3.73	27	12.86
10	4.14	28	13.27
11	4.58	29	13.65
12	5.05	30	13.98
13	5.54	31	14.28
14	6.05	32	14.53
15	6.58	33	14.73
16	7.12	34	14.88
17	7.67	35	14.97
18	8.22	36	15.00

Control Points	Stat Level	Top Speed
MIN	1	2.00
MAX	36	15.00

Figure 20: A table that demonstrates the growth of a stat's value based on the stat's level. The formula uses the min and max value for the stat below, and interpolates across the two values to generate the growth curve.

Above is a table that calculates and displays the top speed stat's growth based on skill level. By entering a minimum and maximum level at the bottom (the stat values for levels 1 and 36 respectively), the table auto-generates the band of values the program comes up with using the blending function selected from the stat growth curves.

The final table in the spreadsheet applies the different stat levels in a series of theoretical situations. The chart below pits a character of mean level X versus a vanilla ship (all stats at level 6) firing at point-blank range to determine best-case time to kill the vanilla ship. By using this chart, stat data can be tested without needing to play the game and test the values, which sped up development and iteration on the stats considerably.

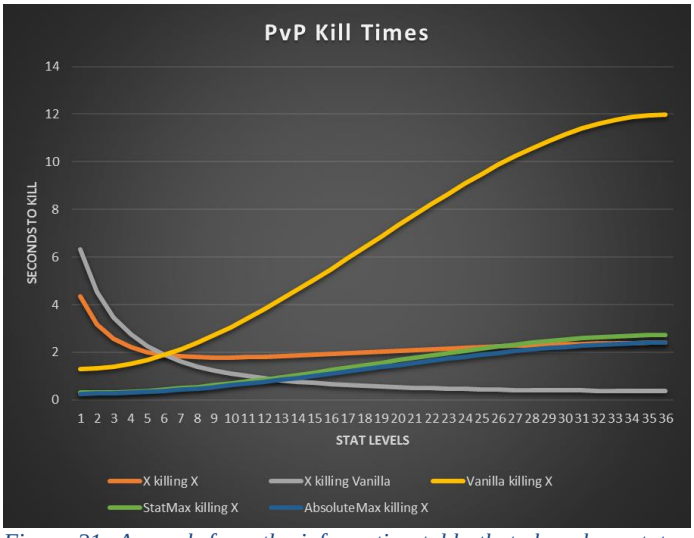

Figure 21: A graph from the information table that show how stats manifest in-game. This table shows how long a player with particular damage and rate of fire levels (X) would take to defeat and be defeated by other players, such as a vanilla player or a player with maxed-out stats (all stats at level 26).

The project employed concentric development to organize the game's components and features into discrete tiers. Each tier built off the previous tier, and provided a clear path for the project's dependencies. The tiers also defined feature priorities

dividing them naturally into milestones. The “foundational” tier consisted of mandatory engine features and bugfixes work to be undertaken before beginning the project. The “stretch” tier was considered optional, and comprised the stretch goals of the project.

The “core gameplay” tier consisted of all core elements that made up the game. These features focused on getting the game functional first, proving out the core activity loop and gameplay elements before moving on to polish tasks. This tier also included multiplayer, player ships and rudimentary enemies, the game’s basic power-ups, and a level in which to fly around. The game flow through the Assembly and Challenge phases was also implemented, along with start and end UI. Most of the content wasn’t polished to final quality, but served as the skeleton for the rest of the game’s features. This tier was similar to a Proof of Concept Gameplay milestone, with an emphasis on playability.

The “feel” tier focused on getting gameplay smooth and polished. This tier was created to mitigate the risk of overscoping up front and expending polish time, so that before any secondary features and functionality were added, the game already felt good. This established the minimum-viable product for the game, and ensured that the project met the goal of creating a complete game.

The “additional content and balance” tier’s tasks focused on augmenting gameplay quality and replayability. This tier introduced equipment, and added the remaining power-up pickups and stats. Completion of the tier’s tasks added 4 more minigames, as well as procedurally generated map zones during the Assembly phase. These features were polished to match the quality of the game after the “feel” tier was finished. Once these tasks were completed, the project was in a state that development could be stopped and the game still felt complete and polished, ready for defense.

All remaining tasks and stretch goals were relegated to the “stretch goals” tier, optional for completion. This tier included features such as Assembly phase bosses and additional minigames. The tier added new equipment variations, such as new weapons and chassis types. This tier pushed the quality bar and polish level of the game, and any remaining content post-defense will be considered future work.

By applying concentric development, the project was not only organized into discrete milestones with clear objectives and deliverables, but was separated into a chain of dependencies that prioritized its core components.

IV. Postmortem

A significant amount of development time was spent working towards the creation of highly-reusable engine systems, with mixed results. Many of these subsystems were attempted in order to expedite future work, but the payoff wasn’t always within the project’s scope. The constant desire to do things the “right way” in an attempt to further demonstrate technical mastery wasted time that could have been put to better use. Because of the somewhat nebulous goal of the project (“demonstrate mastery”), it was easy to lose sight of short term goals while pursuing perfection. After this mistake was made a few times during the artifact’s creation, the developer retargeted towards ensuring that the artifact was finished, as opposed to creating a set of impressive subsystems and an unplayable game. Instead, the game was created with workable systems and some practical “work in progress” solutions. Since anything can be refactored and reworked post-project, the game didn’t need to be architecturally perfect; it just had to work and demonstrate mastery.

For example, time was spent planning, designing, and trying to implement a complex UI engine subsystem that worked within and outside of the SpriteGameRenderer. However, this proved to be a goal that wasn’t worth the amount of effort, in respect to the timeframe of the thesis. In the end, all that was essential was support for text and bar graphs inside the SpriteGameRenderer itself, which was easier to implement, served the immediate needs of the project, and ultimately worked well enough to support the game. This problem helped dispel the myth that only lofty, future-proofed systems are the “right way” to solve engine problems for games. Programming in a custom engine creates a temptation to solve problems the game doesn't have yet. Time constraints help to prevent indulging the temptation, as they force the developers to solve the most urgent problems instead of tackling ones they don't have. Good engineers spend the right amount of energy on the right problems.

Conveyance was another major struggle during the project, and ended up being one of the most important aspects of the game (obvious perhaps in hindsight). Players need to be able to understand the game, and any lazy shortcuts developers take can negatively impact the player experience. For example, the equipment system was confusing and unwieldy throughout most of the project. Whenever players moved over a piece of equipment, it was automatically picked up, causing them to either wonder how they gathered the equipment or to ignore it completely. This not only showed up as a complaint multiple times throughout that period, but distracted from other issues that needed feedback and wasted playtesting time. This remained in the project as a to-do until less than a month out, when it was replaced with a system in which players must hold a button to pick up equipment. Solving the issue sooner (which ended up being only a 5 minute fix) would have gathered better player feedback and created a more positive gameplay experience.

V. Conclusion

This thesis aims to demonstrate an all-around mastery of the lessons and skills developed through the Guildhall’s software development track by creating a well-polished game prototype. By building a multiplayer competitive, open-arena, 2D twin-stick shooter from the ground up, polishing and optimizing the game, the artifact supports the thesis’ claim of mastery.

VI. Future Work

Despite the effort put into the artifact, *AllStar* remains a project, not a product. Several areas of the game would need to be addressed to bring it up to shippable quality. Art assets are either from the public domain or from another artist who had limited time to contribute to on the project. As such, most of the game’s art style is not cohesive, and lacks the level of quality and beauty an indie game would need to succeed on the market today. The game’s design and balance require a few more

iterations as well. Because the project's focus was on creating content and systems to demonstrate mastery of the programming track, less time was spent on the design and balance necessary to bring the game to market. *AllStar* has reached the stage of development where iteration and content creation are much easier to do, which would help speed up development for the remainder of the project.

VII. References

[1] V. Chelaru, "Rock Paper Scissors - A Method for Competitive Game Play Design," 23 January 2007. [Online]. Available: http://www.gamasutra.com/view/feature/130150/rock_paper_scissors__a_method_for_.php. [Accessed 8 September 2016].

[2] R. Terrell, "Shared-Multi-Split Screen Design," 17 June 2011. [Online]. Available: http://www.gamasutra.com/blogs/RichardTerrell/20110617/88846/SharedMultiSplit_Screen_Design.php. [Accessed 8 September 2016].

[3] S. Young, "Good Robot Postmortem #2: Gameplay," 19 July 2016. [Online]. Available: http://www.shamusyoung.com/twentysidedtale/?p=33343. [Accessed 8 September 2016].

[4] J. e. a. Hoffstein, "Group Report: Progression Systems," in *Project Horseshoe*, Comfort, 2014.

[5] Kirby Air Ride. (GameCube). JP: HAL Laboratory, Nintendo, 2003.

[6] Realm of the Mad God. (Adobe Flash). USA: Wild Shadow Studios, Deca Games, 2011.

[7] Sinistar. (Arcade). USA: Williams Electronics Inc., Williams Electronics Inc., 1982.

[8] Galak-Z: The Dimensional. (Microsoft Windows). JP: 17-BIT, 17-BIT, 2015.

VIII. Figures

IX. Appendix: Rendering Pipeline & Profiling

Overview

The SpriteGameRenderer renders various Sprites, *ParticleSystems*, and other objects that extend the *Renderable2D* class. The SpriteGameRenderer allows sprites and other renderables to self-register to various SpriteLayers, which each have an in-place linked list of all the Renderable2Ds on that layer.

```
class Renderable2D
{
public:
    //CONSTRUCTORS/////////////////////////////////////////////////////////////////
    Renderable2D(int orderingLayer = 0, bool isEnabled = true);
    virtual ~Renderable2D();

    //FUNCTIONS////////////////////////////////////////////////////////////////////
    void ChangeLayer(int layer);
    void Enable();
    void Disable();
    virtual void Update(float deltaSeconds) = 0;
    virtual void Render(BufferedMeshRenderer& renderer);
    virtual AABB2 GetBounds();
    virtual bool IsCullable() { return true; };

    //MEMBER VARIABLES/////////////////////////////////////////////////////////////
    Renderable2D* prev; //In-place linked list, points to the previous renderable in the
    Renderable2D* next; //In-place linked list, points to the next renderable in the list
    int m_orderingLayer; //Drawing order is ordered by layer, smallest to largest
    bool m_isEnabled; //If disabled - does not get rendered
    bool m_isDead = false;
    uchar m_viewableBy;
};
```

Figure 23: The header file for the Renderable2D class. The class features previous and next pointers to make each renderable a node in an in-place linked list.

Rendering

The render cycle starts with game code instantiating a subclass of the *Renderable2D* class, which is an abstract class that provides the interface for all renderable objects in the scene. Each Renderable2D has the ability to register and unregister itself from SpriteLayers, as well as update and render functions. The interface also provides functions for grabbing the renderable's bounds and whether or not a particular renderable is cullable or not, both of which are used when determining what to draw onscreen.

Each of the inheriting classes of Renderable2D are for drawing a new type of object. These classes include: *Sprite*, which is used for rendering textured quads to the screen; *TextRenderable2D*, which draws text with kerning support; *BarGraphRenderable2D*, used for drawing bar graphs; and the *ParticleSystem*, which renders particle emitters.

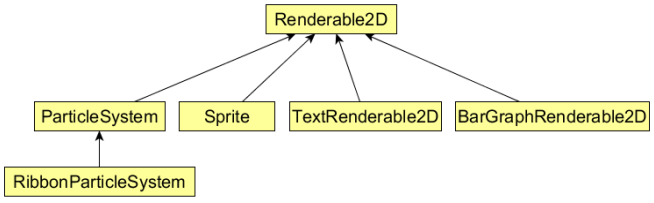

Figure 24: UML diagram of the Renderable2D class hierarchy. Any subclasses of Renderable2D can be auto-registered to one of the SpriteGameRenderer's SpriteLayers.

SpriteLayers are classes that group together renderables via an in-place linked list and render them per frame. Each SpriteLayer is responsible for registering and unregistering sprites, along with holding and applying any *FullScreenEffects*, a container object for an FBO post-process material, when rendering. The SpriteGameRenderer uses these layers to determine the draw order for groups of objects in the scene. Each layer also has a series of controls to toggle bloom, change the layer's virtual scale, or disable culling for that layer.

The SpriteGameRenderer draws all the layers for each of the player viewports registered by the game class. When the number of player viewports changes, the SpriteGameRenderer divides up the screen into the required viewports and creates a series of render targets for each. Since all the viewports are equally sized for a multiplayer game, a pool of 4 textures is created that will fit all the player's views. Once this pool has been created (or reused, if the state hasn't changed), the SpriteGameRenderer proceeds to draw a world view based off the first camera. This render pass includes drawing all the SpriteLayers, along with every registered Renderable2D subclass in the in-place linked list for the layer. During each layer's render, all renderables are checked against the camera's viewport to determine which of the geometry can be culled. All of it is drawn to the viewport-sized canvas, including all FBO effects for that player. Once the player's render pass is complete, this render target is then copied onto another full-screen-sized FBO in the appropriate place. The SpriteGameRenderer then renders the remaining players to the full-screen FBO, which then completes one more full-screen effect pass, allowing for FullScreenEffects that span multiple viewports. Once this step is completed, the whole FBO is copied to the back buffer, and the render process begins again.

Bloom

Because computer monitors have a fixed brightness per pixel, we must fake this brightness via some other means. By applying a Gaussian blur to any bright objects in our scene, the objects will appear brighter within the constraints we have. When rendering the game world, bright sprites (designated by being added to bloom layers) are written out to a 2nd color target when drawn. Then, the software applies a Gaussian blur for several horizontal and vertical passes. The two-pass approach is more efficient than doing both simultaneously, as simultaneously blurring with a 32x32 kernel size takes 1024 samples/fragment versus 64 if done in two passes. Once the 2nd color target has been blurred, the SpriteGameRenderer finishes the bloom effect by compositing that second target with the first, creating a bright-looking laser or explosion.

Profiling

Later into the artifact's development, the author came across a performance issue in the project. While it eventually turned out to be something unrelated to the artifact causing the framerate to drop, profiling not only helped to discover poor rendering practice, but also improved performance on lower-end machines. The developer generated a profiling report based on the most expensive operations based on self-time, or how long a subsection of code took minus the duration of its' children.

```
---===Frame impact report===---
Frame 0's time:      16.79ms
///TOP///
TAG                          NUM CALLS  NUM DRAWS  NUM ALLOCS SIZE ALLOCS   SELF TIME  TOTAL TIME    MAX TIME    AVG TIME PERCENTAGE
RenderFromFBO                      193          0           0          0       0.81ms      0.81ms      0.01ms      0.00ms      4.81%
Mesh Init                          137          0           0          0       0.81ms      0.81ms      0.02ms      0.01ms      4.81%
AssemblyUpdate                       1          0           0          0       0.56ms      0.56ms      0.56ms      0.56ms      3.36%
BindToVAO                          193          0           0          0       0.45ms      0.45ms      0.19ms      0.00ms      2.71%
```

Figure 25: A sample from the profiling report, sorted by the respective self-time for each call.

Although the most impactful on frame time was RenderFromIBO, this segment was based on the number of draw calls the game was making. What was unusual was the MeshInit subsection, as it was taking up a large amount of time comparatively to the rest of the program. Upon closer inspection, this function was generating new render buffers (a VBO and IBO) once every draw call. After a single use of them, they would be discarded and regenerated for the next call. This was addressed by only generating a new render buffer if one didn't exist, and to only destroy the buffers on destruction of the mesh. After these changes were implemented, Mesh Init fell from 0.81ms to 0.31ms, about a 60% reduction in frame cost.

Another optimization attempted was regenerating the VAO bindings whenever the program detected a change, instead of for each draw call. The code was being called 177 times and took 0.16ms, which was reduced to 88 calls and 0.12ms, a 25% reduction in frame time. While this ended up being a relatively insignificant optimization, the refactor helped to broaden the developer's understanding of the rendering pipeline.