Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Game Design and Development

Capstone Final Design & Development Approval Form

Student Name: Jeff Bauer

Student Name: Mark DelFavero

Student Name: Victor Shu

Student Name:

Project Title: Super Collider

Keywords: party, game, ragdoll, casual

Christopher A. Egert, Ph.D.
Lead Capstone Advisor

Erika Mesh
Capstone Process Advisor

Austin Willoughby
Faculty Advisor

Sten McKinzie
Faculty Advisor

Erin Cascioli
Faculty Advisor

David Schwartz, Ph.D.
Director, School of Interactive Games and Media

Super Collider

By

Jeff Bauer

Mark DelFavero

Victor Shu

Project submitted in partial fulfillment of the requirements for the degree of Master of Science in Game Design and Development

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

May 5, 2020

Acknowledgments

We would like to thank Austin Willoughby, Erin Casciolli and Sten McKinzie for working with us on this project as our advisors, as well as all of the IGM faculty. They provided us with a lot of insight and suggestions over the course of the project. We'd also like to thank everyone on our extended team including our modelers, sound designers, and composers. Lastly, we would like to thank all of our playtesters that helped us to test out various mechanics and ideas during development.

Executive Summary

Super Collider is a physics-driven party brawler that was designed to create a wacky and hectic experience for up to four players at once. This genre was chosen because our team is passionate about bringing people together to sit down in front of the same terminal and enjoy a game together. To make this game as good as it could be, we conducted research into physics-driven character animation, visual effects, and randomness. We used Discord, GitHub, and Google Sheets for task management, and communication. Regular code reviews and proper use of Git allowed us to always keep track of what was currently in development. A primary goal of our game design was for each weapon or level to bring something new to the table. We also wanted our weapons, levels and controls to be easy to learn while also having a hectic enough nature that they would encourage players to experiment and innovate during gameplay. We built the game in Unity using C# and C++. The majority of the code is written in C#. C++ was used for low level logic that C# does not easily support. We built custom shaders and implemented a variety of visual effects to give the game the look and feel that we wanted. We worked closely with our artists to ensure that our weapon models were easily recognizable and had the look and feel of prototype weapons that would belong in an experimentation facility. Over the course of development we conducted playtests so that we could get feedback on our work. Our team made acknowledging player and faculty feedback a high priority, and made many changes based on the feedback we were receiving. While we did not meet all of our stretch goals for the project, we feel that the game has come a long way and plan to continue working on it until it is in a publishable state.

Table of Contents

List of Figures

List of Tables

1 Introduction

Super Collider is a competitive game in which players must use objects in their environment to eliminate other players and be the last one standing. The goal for *Super Collider* is to provide players with wacky experiences that facilitate hanging out with friends and low tension competition. This precise experience is fostered in *Super Collider* by focusing on weapons and environmental effects that are specifically tuned to have interesting physics interactions with players and other objects.

Super Collider falls into the physics-based subgenre of the party brawler genre. Our team's interest in exploring the party brawler genre comes from a wide variety of positive experiences individual members have had over many years with this genre. We selected the specific subgenre of physics-based party brawlers because we believe that there is still a lot of untapped potential in this space. The party brawler genre is particularly strong when you have three or more players that want to enjoy a low stakes game together, but falls short with fewer players or in a competitive scene. Properly tuned physics-driven games can have interesting and unique interactions, but also tend to be buggy and cannot easily be networked due to the large amount of data that would need to be synced between players. Since we wanted to focus on friendly matches between players in front of the same terminal, we felt that the genre of physics-based, party brawlers was a good fit for the scope and goals of our project.

The main focus for the research we conducted was improving the experience of our gameplay. We conducted some general research into competitive games in the genre, but our primary focus was our

individual research. Jeff's research focuses on how other games have implemented physics-driven character controllers. Victor's research focuses on the aesthetic goals we had for the project and what kind of shaders and visual effects we could use to achieve them. Mark's research focuses on how to evaluate and describe randomness mechanics in games. These research topics were selected to add value to our game.

This paper will cover the research that we did, the processes that we used, the game mechanics we implemented, the technical details of those implementations, the style of assets we used, the results from our playtesting, and our post mortem.

2 Game Genre Background and Market Analysis

	Max Number of Players	Physics Driven Combat	3D	Item/Pickup Based Combat	Environmental Hazards	Separate Single Player Mode	Number of Maps (Excluding non-versus modes)
Gang Beasts	8	X	X		X	X	20
Stick Fight The Game	4	X		X	X	X	80
Move or Die	4				X		Many
Swordy	8	X	X	X			3
Super Smash Bros. Ultimate	8			X	X	X	108
Rayman M PS2 (Excluding Race Stages)	4		X	X			13
The Legend of Zelda: Four Swords Adventures (Shadow Battle Mode)	4			X	X	X	10
Super Collider	4	X	X	X	X		9

Table 1. Market Analysis

There are many party brawlers out there and it is a fairly well-established genre, but the particular subgenre of physics-driven party brawlers remains mostly unexplored. Physics can be quite expensive computationally and can put design constraints on a game that may clash with the designer's vision. In highly tailored experiences, there is little room for the oddities and chaos that physics can bring into a game's world. The term "physics" is also deceptively broad when discussing games as it is not always limited to kinematic interactions between objects in the game world. Games can also have features such as physically accurate sound, physics-based rendering, or physically animated fluids. The term "physics-driven combat" as used in this paper, refers specifically to combat in which the animations, hitboxes, and movement of actors are based on and driven by kinematics. When using this definition, the subgenre of physics-driven party brawlers becomes much more distinct and identifiable. Games such as *Super Smash Brothers* that clearly feature kinematic physics interactions but do not use kinematics to determine hitboxes or animations, will not be considered as having "physics-driven combat."

Party brawlers have a tendency to use certain features such as items or environmental hazards to make the gameplay more varied. Allowing players to pick up a variety of items forces them to choose between directly confronting an opponent with what they have now or attempting to pick up a more advantageous weapon first. This dilemma makes the gameplay more interesting and varied. Similarly, stage hazards force players to keep their attention divided between their enemies and the level itself. A skilled player might prioritize survival over confrontation, or even use the hazard to their advantage by baiting their opponent into a trap. Items and environmental hazards have both proved to be staples used in such games as *Stick Fight: The Game* and *Super Smash Brothers.*

One of the main goals for *Super Collider* is to have 3D physics-driven combat that also features weapons and stage hazards. The goal for the resulting combat is that it would benefit from both the wacky unpredictability of ragdoll physics and the wide variety of options that weapons can add. While this may sound simple on paper, this approach requires fine-tuning the physics properties of every weapon, projectile, and any possible resulting interaction. While some of the games in the table above use these features, *Super Collider* is intended to stand out due to its usage of weapons and stage hazards in a 3D physics-driven environment.

3 Active Ragdolls

Jeff Bauer

The section describes research into the methods of creating active ragdolls, specifically using the Unity physics engine. Active ragdolls are a method of simulating the movement of characters through the use of forces applied to their skeleton. I was first introduced to the concept of active ragdolls in 2014 through the beta release of Gang Beasts, a silly PvP brawler. Every animation action and animation in Gang Beasts was done procedurally, which I had never seen before then. The game also caught the attention of many other developers, mostly those who used Unity, and games and tech demos began to surface, each with their own implementation of active ragdolls. Active ragdolls were and still very much are in the Wild West. There is currently no "correct" or "best" method for creating these kinds of systems.

I have also been hard-pressed to find any formal documentation or research on this topic in the realm of games. I say specifically in the realm of games because it is obvious that a lot of research has been done for creating physics simulations of skeletal creatures. However, these papers focus primarily on realism - they want to create the most realistic simulation to best recreate the real world. For many developers, realism is not the primary objective when implementing active ragdolls. Instead, it's just enough of an approximation of the real thing to be understood as that real thing, but so far off from the real thing that it becomes goofy and entertaining. When developers do speak up about their implementation of active ragdolls, it is often a very vague and short description about the concept they entailed that is just enough to point you in the right direction of things that you should be looking at, but the entire implementation is still ultimately up to the individual to figure out. My hope is that this research component in combination with our discussion about the character controller in the technical design portion of this document as well

as resources referenced will provide clear enough guidance to where an individual will have a clear idea about how to approach implementing an active ragdoll system in Unity.

I will begin by describing two active ragdoll systems. I treat these systems as polar opposites for how an active ragdoll system can be implemented. Thus, if we understand both, we can decide what elements we want to implement for our active ragdoll system based on our needs. It is certainly not a comprehensive list of all possible methods of implementation, but they give a clear indication of the two roads one can go down.

Figure 1. Active ragdoll system of Gang Beasts

The first implementation I want to look at is used in Gang Beasts. Though we do not have the source code for Gang Beasts, Boneloaf, the developers of the game, have done a number of talks about their system as

well as posted some debug videos of the game. Using this information we can infer and reverse engineer their implementation. Firstly, as can be clearly seen in the debug video, there is a large ball attached to the pelvis of the player. Being the primary contact with the ground, this ball serves as the main method for balancing the player. It is also very easy to move the character around and turn them as all you need to do is move and turn the ball. The feet in Gang Beasts are purely for show and do little for affecting the actual movement of the characters. This design plays well to the gameplay of Gang Beasts, which heavily focus on the upper body with punches and grabs. This design favors a stouter model with short legs so that the invisible sphere holding the character up, which does collide with other surfaces, does not have to extend very far out from the body.

The characters are held upright by an upwards force applied to the head. However, only applying an upwards force to the head will leave the character feeling floaty since this force is constant and in the opposite direction of gravity. To counteract this, an equal but opposite force is applied to the pelvis. This keeps the character always standing upright, but will keep its acceleration due to gravity consistent with any other rigidbody that does not have forces applied to it. This system for keeping objects upright is consistent with a number of implementations that I've run into.

This concept of equal but opposite force extends beyond just balance. In Gang Beasts all appendage movements are done by applying a force on the appendage in the direction it needs to move and then applying an equal but opposite force at another position or on another connected rigidbody in order to keep the whole system from being pulled in the direction of the first force. For example, if we wanted to move an arm to the right in Gang Beasts, we would apply a force to the hand in the right direction and

then an equal force in the opposite direction to possibly the shoulder. The benefit of this sort of implementation is that it is extremely simple. Gang Beasts provides players with the ability to individually move their arms for grabbing and climbing structures, which can easily be done by applying a force to the hands in the direction of input.

Additionally, it is very easy to chain multiple actions together, and the developers of Gang Beasts speak very positively to this. In this talk, they speak to how they implemented 3 basic actions, walking, jumping, and crawling, but did not limit the player to being able to do only one at a time. As a result, players could chain multiple actions together, creating new actions such as rolling, diving, and handstands. This made for some interesting gameplay, as each time a new basic action was implemented, players would gain an exponentially increasing number of possible actions by chaining different basic actions together.

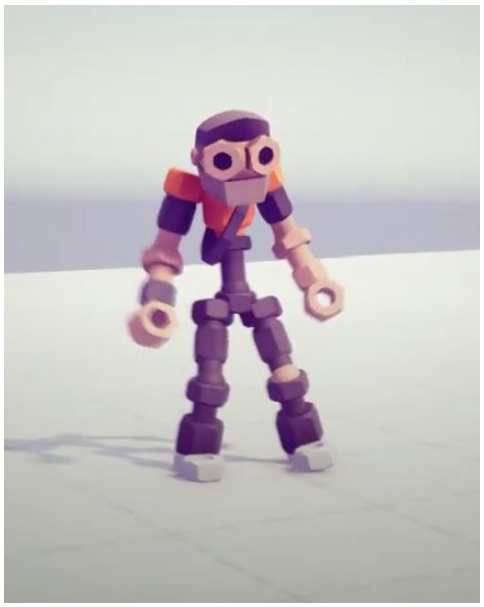

Figure 2. Active ragdoll system from MetallCore999

The second implementation I want to look at is an implementation in an unnamed project created by a person known online as MetallCore999. This implementation takes an approach much closer to simulating how a bipedal character actually works, including how muscles operate and balance the system. Instead of using forces applied to the ends of the arms or legs, MetallCore999's implementation makes use of setting the target rotation of joints, which simulates muscles expanding and contracting. Instead of a ball holding the character's body up, the body is held up by the character's legs. Without the invisible sphere, the legs can be as long as the designer chooses, and having longer legs makes the process of balancing easier.

Since there is no invisible sphere to push around, the only forces available to move the character is the friction between the feet and the ground. By pushing backwards with the feet, assuming the friction between the feet and the ground is strong enough, the character will be propelled forwards. The feet are pushed back by rotating the joint connecting the thigh to the pelvis while straightening the knee.

Using this basic information about how some active ragdoll implementations work, we can begin looking at how one might begin to implement this in an engine. Since *Super Collider* was built in Unity, I will talk about how one could begin to approach this in Unity. Nimso Ny (https://twitter.com/nimSony) is a game developer who has done a lot of work in the spaces of VR and physically based character controllers within the Unity engine. In a reddit post, he pointed out some of the key things one will need to understand when building an active ragdoll in Unity. Firstly, active ragdolls are built using rigidbodies connected by joints. You can use the ragdoll builder that comes standard with Unity or you can build one yourself from scratch. When building your ragdoll, make sure you are using Configurable Joint

components. Other components like the Hinge Joint, Character Joint, and Spring Joint may be nice for testing things, but you'll ultimately want to use Configurable Joints for everything since they give you the most control over your joints. Specifically, Configurable Joints are the only joint that allows you to make a ball and socket joint (very useful for characters) with a single joint component.

Secondly, every joint should have some strength to it. Here, he is speaking to the target rotation parameters of the Configurable Joint component. Here, you can set what angle you want the joint to target. Keep in mind that this target rotation is relative to the starting rotation of the joint, so the rotation of the joint will always be the identity quaternion when it is first initialized. You will also need to set some strength for the joint, which changes how it will approach the target rotation. Adjust the Angular X and YZ Drives to change how the joint will approach the target rotation. Increase the position spring of each drive to increase the strength or torque of the spring, making it reach its target rotation more quickly. Increase the position damper of each drive to dampen the speed of the joint as it reaches its target. If you target angular velocity is 0 (which it will be in most cases), the will slow down the joint over time as it reaches its target rotation, preventing the joint from endlessly oscillating as it overshoots the target rotation.

Finally, don't be afraid to fake things. For most implementations of active ragdolls, realism is not the goal. These implementations are meant to be goofy. If there is a hack you can do outside of just using the target rotation elements of the configurable joints, and it looks good and doesn't hinder development of the other portions of the controller, by all means go for it. It's all about how the controller ultimately moves and feels that's important and if you need to sacrifice physical accuracy for a good feeling game it

is worth it. Also, above all, don't be afraid to experiment. Active ragdolls do have elements of programming in them, but they're more about design and number tuning. You may have the perfect code for developing an active ragdoll, but you will still need to do a lot of tuning of the numbers to get the controller to feel just right. It is an experimental process and only practising it can help you get better at making these systems.

4 Flat-Lit Shading

Victor Shu

This section describes research on the aesthetic of the game. We decided that a simple flat-lit looking fits our game best, thus we need a non-photorealistic rendering (NPR) technique. Many games use NPR, but unlike physically based rendering (PBR), there is no single standard to follow. The visuals of NPR games vary from each other and produce different feelings. I did thorough research on the approaches of how other games use NPR, and then found the best way to put it in our game.

The first game that came into my mind is *Untitled Goose Game* which was just released back in the days.

Figure 3. A screenshot of Untitled Goose Game

Untitled Goose Game has a very simple and clean look. There is no obvious lighting in the game: the shadows are subtle, and almost all things are made up of large areas of solid colors. It successfully created a 2D feeling with flat shading even though it is a 3D game. Still, there is a little bit of ambient occlusion if we look at the screenshot carefully. The ambient occlusion is also very subtle, but it does make the whole visual style look better.

Genital Jousting is also a great reference game not only for shading and lighting but also for colors and shapes.

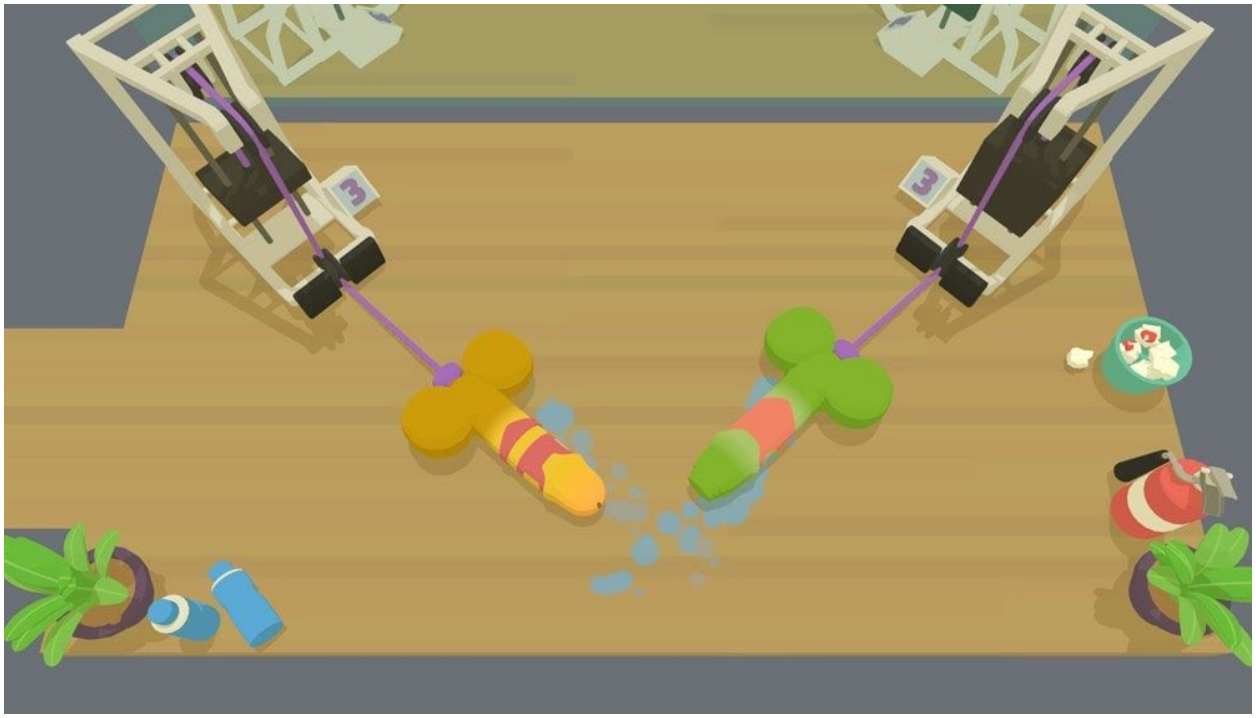

Figure 4. A screenshot of Genital Jousting

The lighting in *Genital Jousting* is much more obvious than *Untitled Goose Game*. To be more specific, the shadows are much clearer and there is a hard edge between the lit part and the dark part of an object. At first sight, the graphics of *Genital Jousting* are more complex than *Untitled Goose Game*, but the compositions are the same: large areas of solid colors. The color choice is also a very good reference for our game. Most of the scenes have a less saturated color while the characters and interactable are more saturated. This makes it feel colorful, but at the same time, it won't become a mess.

Beyond that, I also looked at *Guilty Gear Xrd*, a Japanese anime-style fighting game, which is nearly state-of-the-art regarding NPR.

Figure 5. A screenshot of Guilty Gear Xrd Revelator

As a Japanese anime-style 3D game, there are many more things to consider other than lighting and coloring. According to the GDC talk, the game uses cel-shading, which is basically telling the lit area and dark area apart with a hard edge. There is a threshold to control the size of the lit and dark area. To make the character look 2D under their shader, they control all the factors (threshold, light vector, normal vector) that affect the final result, saving the maps either in the vertex data or in texture.

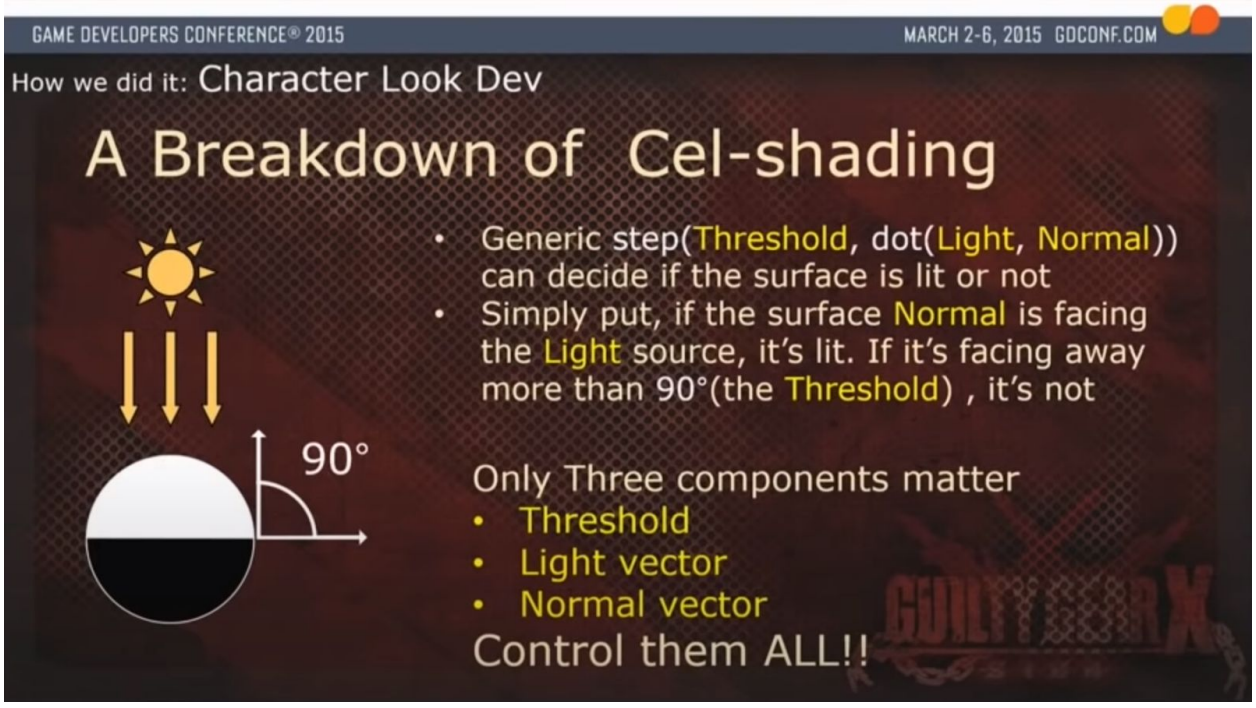

Figure 6. Cel-shading of Guilty Gear Xrd

Figure 7. Comparison between unmodified normals (a) and modified normals (b)

Apart from that, there are many other topics like animation, rigging, and outlines in the talk, but none of these topics are related to our game. And since we are not a big team and don't have people who can work as a technical artist, we cannot directly adapt their methods and workflow.

The first version of the shader used the simplest approach for cel-shading which is controlling the lighting with a threshold. In this implementation, we can also define the shadow color of the material to slightly change how it looks. It looks simple and clean in our game, and we kept this implementation for a while.

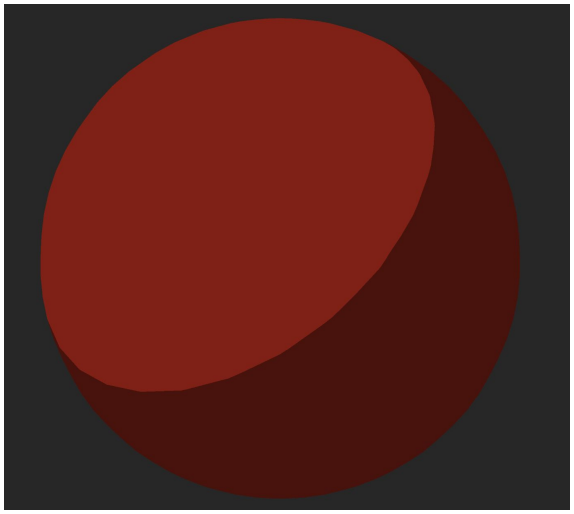

Figure 8. Material of custom shader version 1

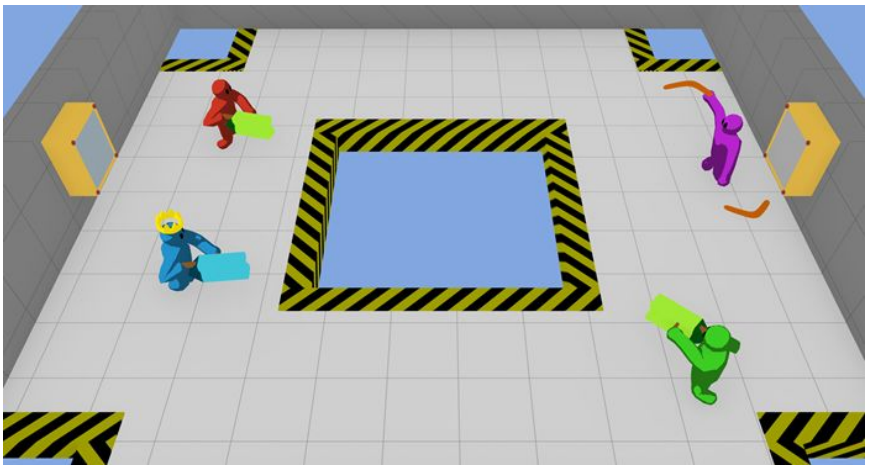

Figure 9. A screenshot of the game using the version 1 shader

The second version of our custom shader took a harder approach. Instead of using a threshold to decide the lit and dark part, I used three window functions to describe the lighting, where each of the window

functions was generated by subtracting two sigmoid functions in regards to the Lambertian reflectance, which is the dot product of the normal vector and the light vector. The shape of the sigmoid functions can be easily adjusted by modifying a pre-defined factor, which can make the transition between two lit parts faster or slower, serving as a parameter that describes the smoothness of the material.

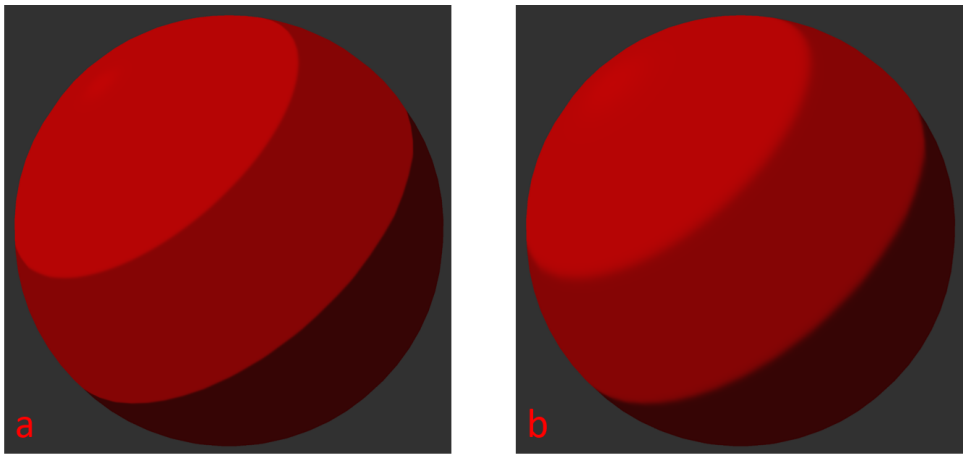

Figure 10. Comparison between smooth (a) and rough (b) material with version 2 shader

I also added a GGX specular highlight to the shader, since it is much more obvious for one to determine the smoothness of the material with specular highlight. The GGX highlight follows the standard BRDF function and is physically accurate. It will also be affected by the smoothness parameter mentioned above.

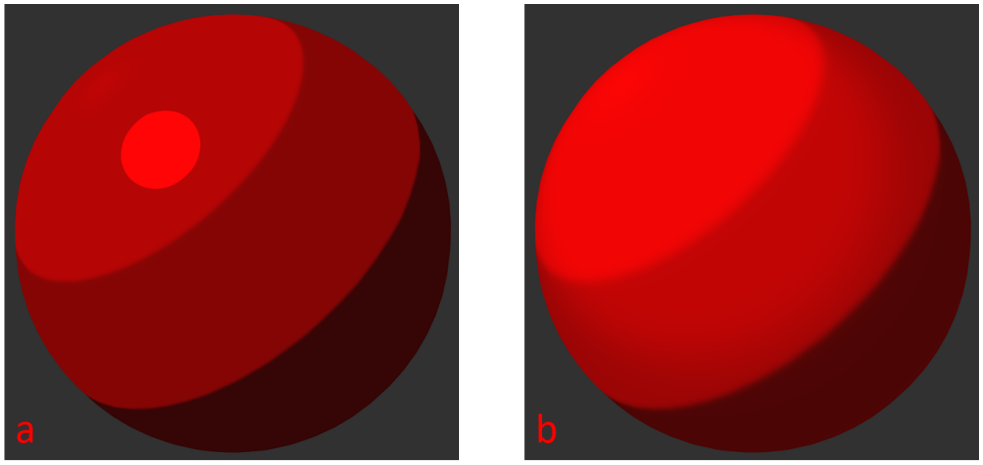

Figure 11. Comparison between smooth (a) and rough (b) material with GGX highlight

In addition, to make it even more physically accurate, I added a fresnel factor to the shader, which basically generates a subtle highlight around the edge of the material.

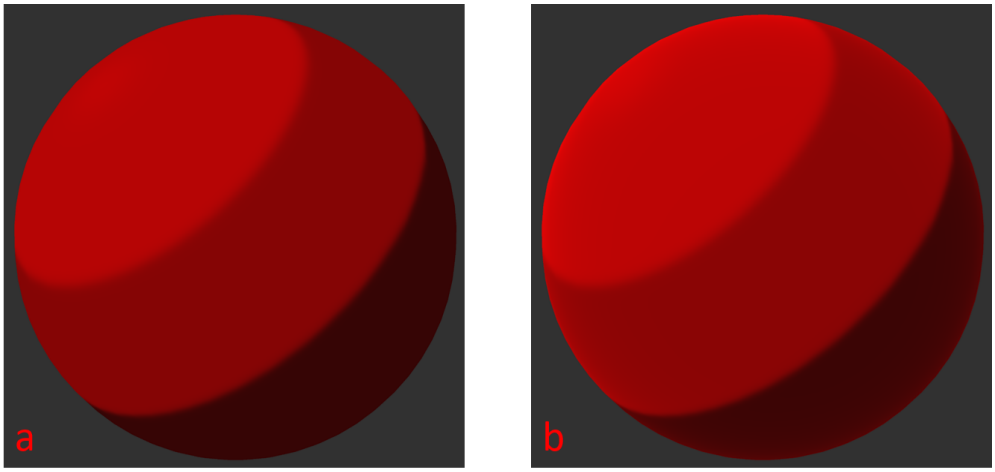

Figure 12. Comparison between materials with Fresnel turned off (a) and turned on (b) of version 2 shader

This version is more or less an experiment to see how good PBR can work with NPR. It was created during the time we are not sure what we want for the visual of the game. However, when we put it into our game, we found out that it made the game look much more complex and lost the simplicity we wanted. Though there are many techniques used in this shader, we decided to not have it since it doesn't fit our game.

We then noticed that we still want the simplicity of the original cel-shading, but there are problems with it. First, since it uses a threshold to control the lighting, there are many cases when two faces of a cube are the same color, which makes it hard for us to see a corner of it. It also reduces the curvature of a round object, mostly our character, making the whole scene look less lively. We noticed that we both want a simple solution but also not an immediate transition between lit and dark parts. So we decided to go back with the old school Lambertian lighting.

The original Lambertian reflectance creates a slower transition than our expectations, but it can preserve the curvature of the object better, like Figure 13 (a) shows. We used the square root of the Lambertian reflectance to create a faster transition, shown in Figure 13 (b).

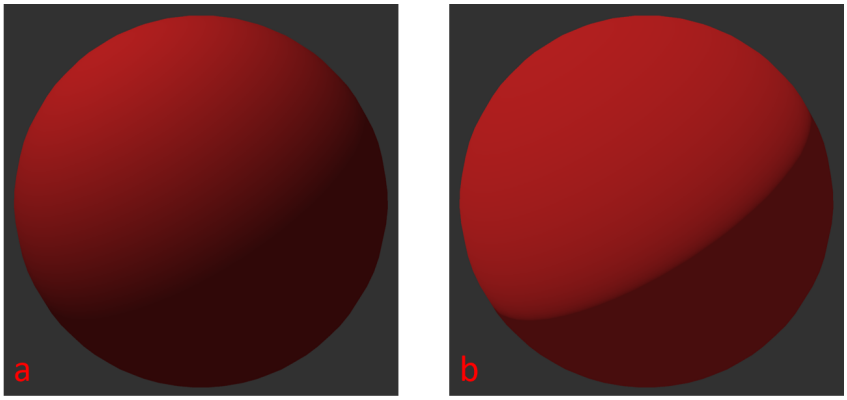

Figure 13. Comparison between the original Lambertian reflectance (a) and our modified one (b)

After putting this shader into our game, we found out that it provides a better 3D feeling and solves the problem of merged faces of cubes. The objects can be distinguished more easily and the curvature of the objects is better preserved, shown in Figure 14.

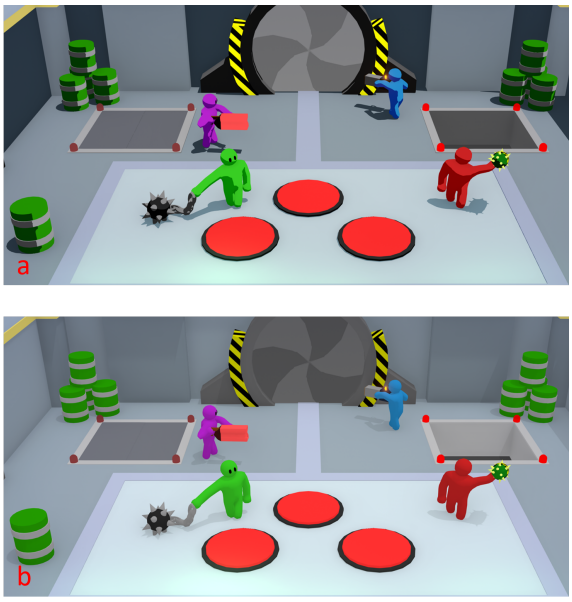

Figure 14. Comparison between the game using version 1 shader (a) and version 3 shader (b)

I also researched some post-processing effects that we could potentially put into our game. SSAO and bloom are the two major effects I looked into. The ambient occlusion will add more feeling of hierarchy to the game, and the bloom effect can highlight objects that we want to emphasize. Figure 15 shows the same scene with both effects off (a), only SSAO on (b), and both effects on (c).

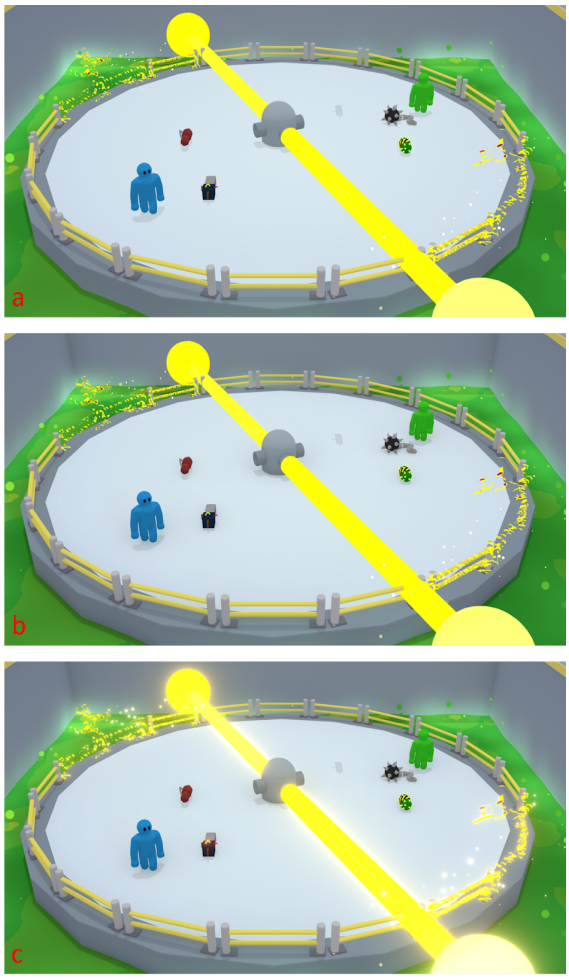

Figure 15. Screenshots of the game with all post-processing effects off (a), only SSAO on (b), and all effects on (c)

5 Taxonomy on Randomness

Mark DelFavero

One of the primary goals for *Super Collider*'s gameplay was for it to create a volatile, messy experience where physics reactions and chance played a significant role in deciding the victor of any given match. As a result, many randomness mechanics are used throughout the game. Randomness in games can have effects including increasing the variety of experiences, protecting player's egos, and broadening audiences (Garfield 2012). In order to better implement randomness mechanics in *Super Collider*, I prepared a taxonomy that classifies randomness mechanics based on the degree at which they are voluntary, signaled, and mitigable. For each attribute, I will define it, evaluate some pros and cons, and examine specific cases.

Some random mechanics are triggered by direct, explicit player action while others are not. A voluntary mechanic can be considered one that the player chooses to activate. By contrast, an involuntary mechanic is one that activates regardless of the player's actions or because of an action that the player is forced to take in order to complete their objective. Mechanics that take place during game set up without player input such as shuffling a deck or generating the game world are also considered involuntary. A hypothetical example of a completely voluntary mechanic, would be one that does not interfere with progression in the game, offers no benefits and completion of which is not tracked by any in game achievement system. An example of a completely involuntary mechanic would be a mechanic that activates automatically, and no amount of player action could possibly prevent it from occurring. Most mechanics do not fall on either of the extreme ends of this spectrum.

To give an example, in *World of Warcraft*, Chromaggus acts as the seventh boss of Black Wing Lair and will randomly have access to two of his five breath attacks each time he is fought. While nothing in *World of Warcraft* forces a player to enter Black Wing Lair, for a player that has decided to complete the instance, the random breath mechanic is considered involuntary because the player cannot opt out of it without opting out Black Wing Lair as a whole. By contrast, the chance to trip while running in *Super Smash Bros. Brawl* would be considered voluntary. Whenever a player is running in brawl, there is a chance that they will trip becoming momentarily immobilized. Since a player can only trip when they decide to use their ability to run and opting out of running does not prevent you from otherwise achieving in game objectives, the chance to trip when running can be considered a voluntary mechanic. There can be some grey areas, such as in *Pokémon Red Version* where random encounters have a chance to occur while walking through tall grass. For the mechanic to activate, a player must actively walk into a tile of tall grass which would typically be considered a voluntary action. However, because the game requires the player to walk through tall grass in order to progress, it can also be considered a forced action making it involuntary. In order to determine how voluntary the tall grass mechanic is in each case, we need to consider the context surrounding that case. If the player's current in game goal is to beat the next gym leader and they cannot reach the next area without crossing tall grass, then the tall grass mechanic is involuntary. However, if an alternate route exists that does not cross the tall grass, then crossing the tall grass is a voluntary mechanic.

Voluntary randomness mechanics are a good way to introduce a dilemma to the player, forcing them to assess the risks and rewards associated with the mechanic. One major drawback of voluntary mechanics is that you risk players deciding that the mechanic is never worth the risk, and wasting the effort put into implementing it. On the flip side, if a player is attempting to one hundred percent the game, they might

feel forced to interact with an otherwise voluntary randomness mechanic in a frustrating way. One potential example of this is in *Yakuza Kiwami 2,* where players need to complete various objectives to complete the completion list, a type of in game accomplishment tracker. One of these objectives is to consume all the different food and drink items in the game at least once, including fifteen different flavors of mystery drink. The presence of the completion list incentivizes players to attempt to drink all the randomly selected mystery drink flavors, but there is no way to influence which flavor the player will get. Involuntary randomness mechanics are a good way to put pressure on a player as they will need to anticipate the mechanics and prepare for a variety of potential results. However, if an involuntary randomness mechanic is too impactful, it may put too much pressure on a player and lead to them opting out of the game entirely. For a new player, it can also be unclear whether an involuntary randomness mechanic is present at all. The first time a player encounters Chromaggus, they might simply think he has two breath abilities rather than that two were selected from a pool of five. This can create interesting experiences or lead to large amounts of frustration.

Some random events provide signals for the player while others do not. A randomness mechanic can be considered signaled if the player is given information about its result before the result takes place. Signals can, but do not necessarily provide the player with the ability to avoid the outcome. The result of a random event can be considered a very subtle, inexplicit signal for what kind of results can occur, but only within the context of multiple iterations. For random events that occur once per playthrough, there might not be signaling except within the context of multiple playthroughs. An unobvious or obscure signal is still a signal and we need to be able to consider certain mechanics as more heavily or more lightly signaled rather than a binary comparison. An example of a completely signaled randomness mechanic would be if the game rolled a six sided die that had six colors on it that would each result in a

different encounter that the player would have to complete, and before the game rolled the die in front of the player, the game showed the player a tutorial explaining in detail what each possible result is and which color indicates the result. An example of a randomness mechanic completely without signaling within the context of a single playthrough would be if the game randomly made the player fight against one of three bosses in an arena without being given any information on the bosses prior to the event, and without having a cutscene before the fight showcasing the opponent.

One example of signaling can be seen in *Slay the Spire*, where players are shown each enemy's intent during the turn before the enemy performs the action. This allows the player to plan out their actions based on the random, but known, enemy behaviors. An example of a signaled randomness mechanic where the outcome is fixed when the signal is observed is the summoning system in *Fate/Grand Order*. Summoning is a mechanic in which players spend the game's primary currency to acquire new characters and equipment. The summoning is accompanied by an animation, and the animation will be different based on the summoned unit's class and rarity. Sometimes, an animation for a lower rarity unit will play at first, but the animation will change part way through to reveal the actual rarity. At the end of the animation, the unit's identity will be revealed, and it will be added to the player's inventory. This signal is used to generate tension and excitement but does not provide the player with any degree of mitigation. This is also a case where the signal attached to the randomness mechanic can be misleading or unreliable. An example of a mechanic without signaling is the loot system used in the *Borderlands* franchise. Players can obtain weapons and skins by killing enemies or looting chests. Some enemies have a higher drop chance for certain items than others, but there is no indication what will drop from an enemy until the player has killed it. World generation in the roguelike *Powder* contains subtle signals. This is because

there are certain patterns to the procedurally generated dungeon that act as signaling to an experienced, observant player.

Signals can be a good tool for setting expectations, increasing levels of excitement, and making the game world feel more consistent. They can give the player time to prepare for the results of a random mechanic and looking for signals can be an entertaining side show. In cases such as the enemy intent system in *Slay the Spire,* signals can play an integral part in the core gameplay loop as explicit, obvious signs. Heavily signaled random events serve to add variety and looseness to the game rather than the unpredictability that random events typically provide. Since the results of previous iterations of the mechanic can be considered as inexplicit signaling, there are no random events without any signaling within the context of multiple playthroughs. However, in cases where the signaling is too complicated to decipher, such as the placement of mining nodes during world generation in *Minecraft,* the signaling for the results of the randomness mechanic can be considered effectively nonexistent. This sort of system promotes exploration and intrigue due to the invariable and unpredictable nature of the results. Signaling can be a powerful tool for defanging certain randomness mechanics or building anticipation before the reveal of a result.

Certain mechanics in games provide players with a chance to optimize their interaction with them, while others do not. A randomness mechanic can be considered mitigable to varying degrees if a player can optimize, escape, mitigate, or recover from the results of the randomly decided aspect of the mechanic. An unmitigable randomness mechanic is one in which the result cannot be escaped or mitigated without opting out of the mechanic entirely. It is an important distinction that it is the result that is mitigable, rather than the mechanic itself. If a mechanic can be avoided but failing to avoid it produces an

inescapable result, then the mechanic is unmitigable. Since a random mechanic can produce multiple results, each result needs to be considered when evaluating the degree by which a mechanic can be mitigated. A hypothetical example of a completely mitigable mechanic would be if a player had to spin a wheel with several potential results on it, but afterwards was given the option to change their result to any of the potential outcomes. An example of a completely unmitigable effect would be if the game spun that same wheel but it decided on one outcome without any player input.

An example of a highly mitigable randomness mechanic is getting invaded by another player in *Dark Souls*. Players in *Dark Souls* can invade each other's single player playthrough to do battle and obtain resources. However, at any time, the invaded host can escape the event by using their black separation crystal to send the other player back to their own version of the world, ending the networked session prematurely. Even though the random matchmaking system placed the other player in their world, they were able to escape the situation. Even if they fight the invader, they can still mitigate the potential bad results of the invasion by skillfully defeating their opponent while minimizing the damage they sustain. By contrast, in *Pokémon Red Version*, when a pokémon attacks another pokémon, the result can be a miss, a hit or a crit. This result is unmitigable because no amount of player skill or player action can possibly change the result of the randomly generated effect. An example that is partially mitigable is if a goblin army invades the player's world in *Terraria*. Once the random event is activated, goblins will continue to spawn until enough of them are killed to clear the event. Since the only way to end the event is to participate, it is inescapable. However, since player performance and skill can be used to minimize the damage a player takes and optimize the benefits from the event, the event is considered mitigable.

Mitigable randomness mechanics provide players with an unexpected opportunity to demonstrate their skill or discretion by optimally reacting to the mechanic. This can improve player experience and create memorable moments. Unmitigable randomness mechanics can add pressure to a player's experience, making them feel the weight of the risk involved in the mechanic. Players aware of a random event that could cause an unmitigated resource loss, will want to ration their resources more carefully so that they can survive the result of the mechanic.

Alternative ways of evaluating randomness mechanics exist and are worth mentioning for comparison. One distinction that can be made is between input randomness and output randomness. Keith Burgun defines input randomness as a "type of randomness [that] informs the player before he makes his decision" and output randomness as "noise injected between the player's decision and the outcome" (Burgun 2014). In terms of the model discussed here, any mechanic that is mitigatable would be considered input randomness, while any other mechanic would be considered as being output randomness. While the degree of mitigation is a useful distinction, without considering other factors, I do not feel it is adequate to explain the difference between randomness mechanics. Burgon also notes that "input randomness, when put up close enough to the player so that he can't plan around it, is basically output randomness" (Burgon 2014). This muddles the definitions a bit, and I think that being able to differentiate between input randomness with and without an adequate level of signaling improves the usefulness of the model.

These criteria are a useful way of categorizing randomness mechanics to help predict the result that the addition of the mechanic will have on a game's gameplay. In the case of *Super Collider*, there are quite a

few randomness mechanics that were implemented that can be evaluated on this scale. Random item spawning in *Super Collider* is an involuntary mechanic with small amounts of signaling and is somewhat mitigable. The ways that items spawn into any given stage are fixed and signals exist to show players where and when items will spawn on most levels. The exact item that will spawn is kept uncertain. Players can mitigate item spawning by using the weapon or preventing their opponent from acquiring it. This combination of attributes leads to item spawning that is somewhat predictable, but because the precise weapon spawned is not signaled, the mechanic produces uncertainty and excitement. It also promotes players to use a variety of items and creates item choice dilemmas as emergent properties during gameplay. Random stage selection is an involuntary mechanic without signaling that is mitigable by optimizing player actions after the result. This mechanic is considered optimizable because even though the player cannot escape the random selection of a level, they can still make the most of it by skillfully playing the level. Random obstacle spawning on the conveyor level is considered an involuntary mechanic with some signaling that is mitigable. These obstacles spawn in a predictable time and place, the only thing without signaling is which type of obstacle will spawn in each spawn location. Players generally have enough time to react to the position of obstacles, but it is possible that the obstacles could randomly form a situation where all the players will certainly die. Even so, that result can be mitigated by attempting to be the last player to die. One conveyor belt obstacle that merits its own description is the laser shooting obstacle. The laser shooting obstacle signals its random orientation with a visual effect informing players where the danger area will be in advance of when the laser fires. This makes it a much more highly signaled mechanic than the other conveyor belt obstacles. One conveyor belt obstacle that was removed from *Super Collider* was a button with random effects. The button had a voluntary randomness mechanic without signaling that was typically unmitigable. The unmitigable effects include launching the player to their death and spawning confetti. The only mitigable effect was that the button would float while it was held down. A skilled player could use this to avoid other obstacles by keeping

their balance, making it an optimizable mechanic. In testing, we found that this mechanic was too risky and that once players knew that one of the results was unmitigable death, they opted out of pressing the buttons. This was considered an undesired reaction, and the random effect buttons were removed from the game.

The degree to which a randomness mechanic is voluntary, signaled and mitigable can play a large part in the type of reaction players will have to it. Voluntary mechanics are an easy way to provide a player with a dilemma. Signaled events give a player the chance to react and adjust their plan before a potential threat or opportunity appears. Mitigable mechanics give players an opportunity to demonstrate their skill and discretion. This framework can be used to categorize randomness mechanics in an effective way allowing developers to predict how they will impact gameplay.

6 Game Development Process

The core team consists of Jeff Bauer, Mark DelFavero, and Victor Shu. Jeff is the administrative and creative lead. All of the core team members have a hybrid programmer and designer role, with Victor leaning more towards engine and user interface work while Mark and Jeff lean more towards game design and gameplay programming. The core team used a task sheet, GitHub issues, pull requests, and Discord to coordinate work on various tasks and features over the course of the project.

The external art team met with the core team regularly to see progress on the gameplay and to coordinate on which models and assets needed to be prioritized for any given milestone. The art team includes Mert Aslandogan, Will Jarrett, and Savvy Blaum. The art team worked on models and other visual art. Aneli Campos worked on sound effects and ambient noise. Dallas Taylor composed our soundtrack. The external art team is also one of the first groups to be introduced to any new ideas and features.

The core team's milestone process was simple and effective. After completing a milestone, the team meets to discuss when the next milestone needs to be, what the goals for the next milestone should be, what work from the last milestone still needs to be done, and whether progress is on track to meet their next milestone. Most of the goals are either feature implementations or bug fixes. All tasks for a milestone are recorded on a spreadsheet and marked as "Not Started." The general process for working on a feature is to mark it as "In Progress" on the task spreadsheet, create a feature branch, implement the feature on that branch, test the feature, open a pull request using GitHub, respond to any comments or critique on the

pull request, merge the pull request after approval, and mark the feature as “Completed” on the spreadsheet. In order to pass review, a pull request needs to have no requested changes and at least one approval including approval from the code owner for the affected section of the project. The code owner for every section of code was the team lead, Jeff. Working on a bugfix is a similar process, except instead of adding a task to the spreadsheet, team members are expected to open an issue on the GitHub project explaining the bug and any observable causes. The bug would either be taken up by the person who found it or the person most familiar with that section of code. After which the process for fixing a bug is to open a feature branch specifically for that bug, fix the bug on that branch, open a pull request that links to the issue it addresses, respond to feedback on the pull request, merge the pull request after approval, and mark the issue as resolved. This process allows for team members to test any upcoming features or bug fixes before they get on the master branch, converse about any concerns they have with the code or implementation, and to better keep track of which changes are currently on the master branch.

Some types of features had additional procedures that were expected to be done before they would be considered completed. Specifically, level development is considered a three-phase process that should include at least two pull requests and two playtests. The three phases in order are the layout phase, the mechanics phase, and the polish phase. The layout phase is completed by creating a rough outline of the level and a brief, informal description that you can use to explain it to other developers. The mechanics phase requires developers to prototype the level in Unity with the mechanics of the level in place, conduct a playtest, and open a pull request seeking approval for the mechanics and layout of the level. The pull request will focus on a variety of questions such as, “Is this level fun?” or “Does this level offer a different experience than other levels?” The polish phase requires developers to make the level feel visually complete, to conduct another playtest, and to open another pull request. This pull request will

focus on visual polish and whether the completed level is good enough to go in the final game. If any issues with the mechanics or layout have been found since the last phase passed, they should also be addressed in this pull request. After the polished version of the level is on the master branch, the future polish will be its own feature and will no longer be tied to the full level design process.

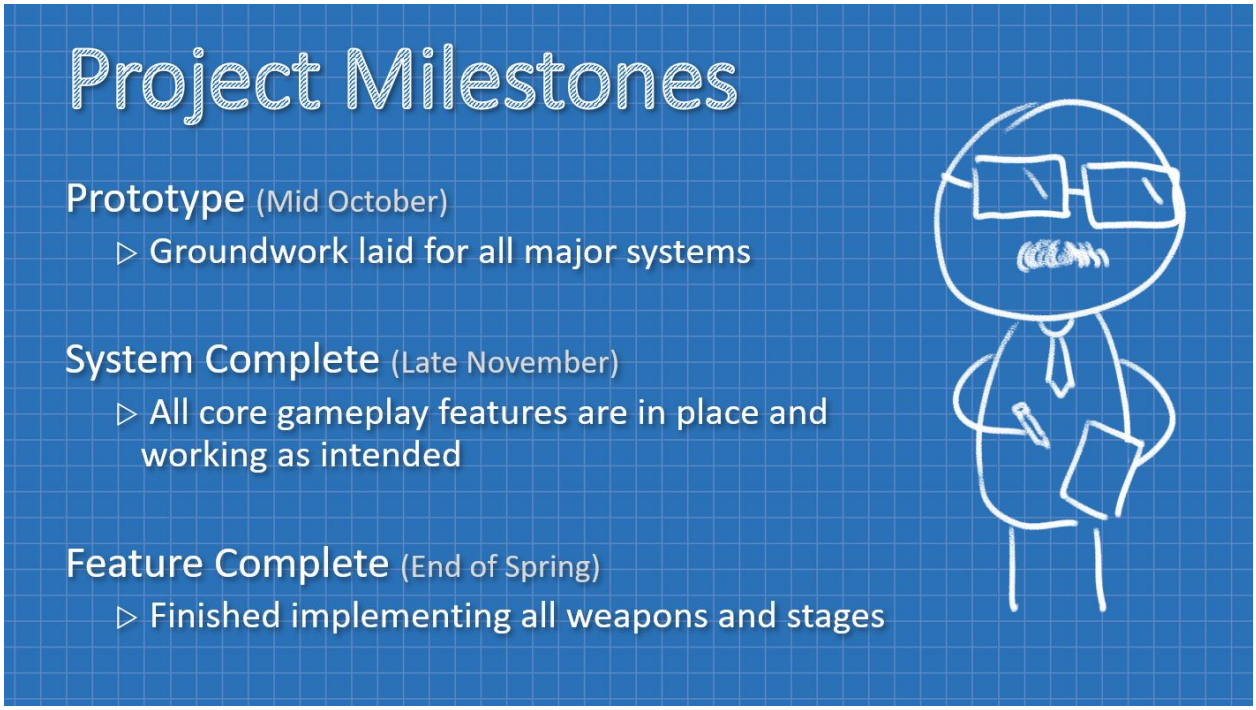

Figure 16. Project milestones set at the start of the first semester

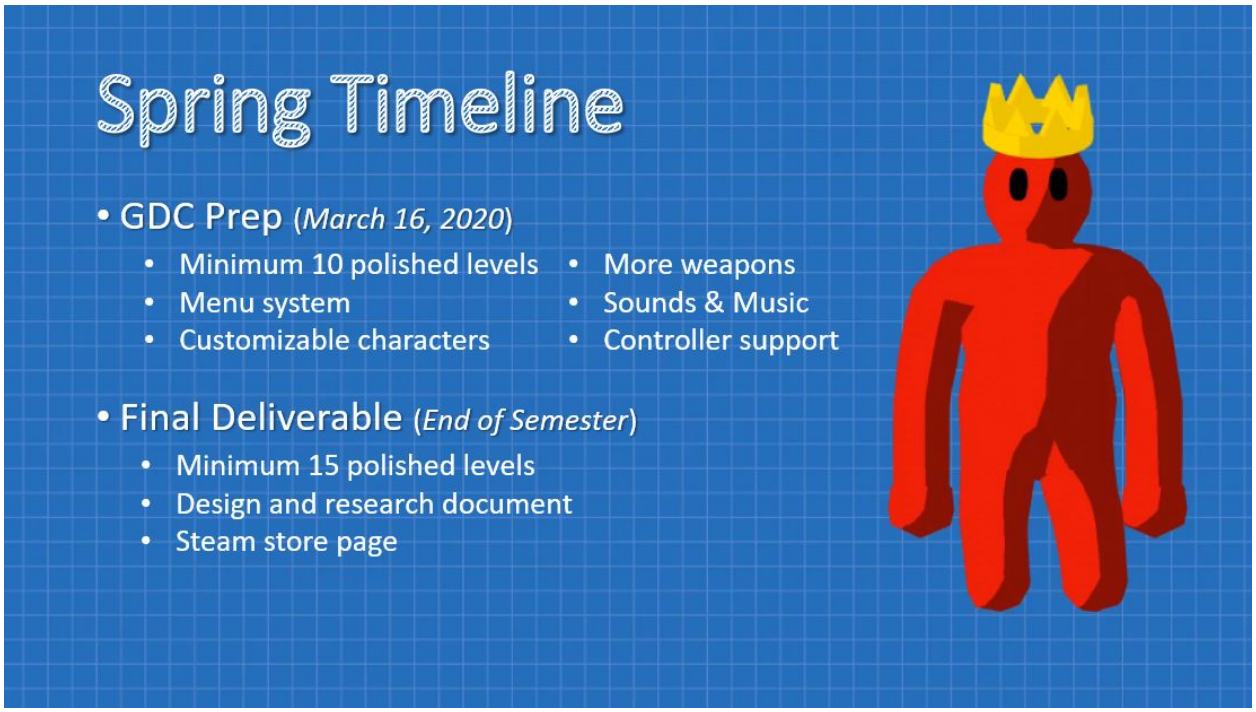

Figure 17. Timeline set for the second semester at the end of the first semester

The team's first milestone schedule was very brief focusing only on major milestones. The spring semester timeline looks very similar, once again bisecting the semester, but with much more specificity on which features would be done by each milestone. Outside of the stated schedule, the team also had minor milestones that were typically scheduled based on playtesting opportunities as they became available.

7 Game Design

Super Collider is all about beating people up in fun, physics-driven ways. A single match features two to four players competing to be the last one standing until a single player has won five rounds. Players can win by using weapons and the environment to devastate their opponents. After a player has won the match, the game is considered over and players are returned to the menu. Each round takes place on a randomly selected level from a fixed list.

The control scheme for *Super Collider* facilitates basic character movement, jumping, picking up and throwing items, attacking with the currently held weapon, and performing a punch when unarmed. This precise set of actions is used to give players choices in their movement and offensive capabilities. Even though the combat focuses on the usage of weapons, players that don't currently have a weapon are still capable of defeating players with weapons. Jumping may seem like a small feature, but it goes a long way to adding maneuverability in a 3D space.

Weapon design is a very important aspect of *Super Collider*. Each weapon needs to have its own pros and cons to usage and have a unique feel to it. Generally, weapons in *Super Collider* fall into one of three categories: guns, explosives, and melee weapons. Guns can be described as any weapon that fires a projectile that is a separate object from the weapon itself. Examples of guns in *Super Collider* are the rail gun, the bomb gun, the ricocheting bullet gun, and the laser cannon. Explosives are one-use items that destroy themselves and cause an effect on surrounding players. Examples of explosives are the

KABOOMerang, the cactus grenade, and the proximity mine. Melee weapons include any weapon that requires that a player approaches their enemy to attack them with it in melee range. Examples of melee weapons include the flail, the rocket hammer, and the rocket glove. All of these different weapons provide players with different angles for an attack. The rocket hammer and the rocket glove give players extra mobility with their speed boost on use. The cactus grenade allows players to limit each other's mobility with an AOE glue effect. The proximity mine is ideal for placing a trap that can catch an opponent off guard. The KABOOMerang is a high risk, high reward option with its powerful offensive capabilities. The flail allows a skilled player to maneuver its ball and chain with various physics interactions, allowing for many angles of attack. The bomb gun fires an explosive projectile that packs a serious punch. The ricocheting bullet gun fires high-speed projectiles that become more powerful each time they bounce allowing players to bounce its projectiles off of various pieces of terrain.

Stage hazards also play a significant role in the gameplay of *Super Collider*. Since players do not have health bars, instant kill stage hazards serve a vital role in the core game loop. Hazards can take the form of acid pits, grinders, portals, exploding barrels, conveyor belts, and lasers. Having a variety of hazards helps create variety within the level design.

8 Technical Design

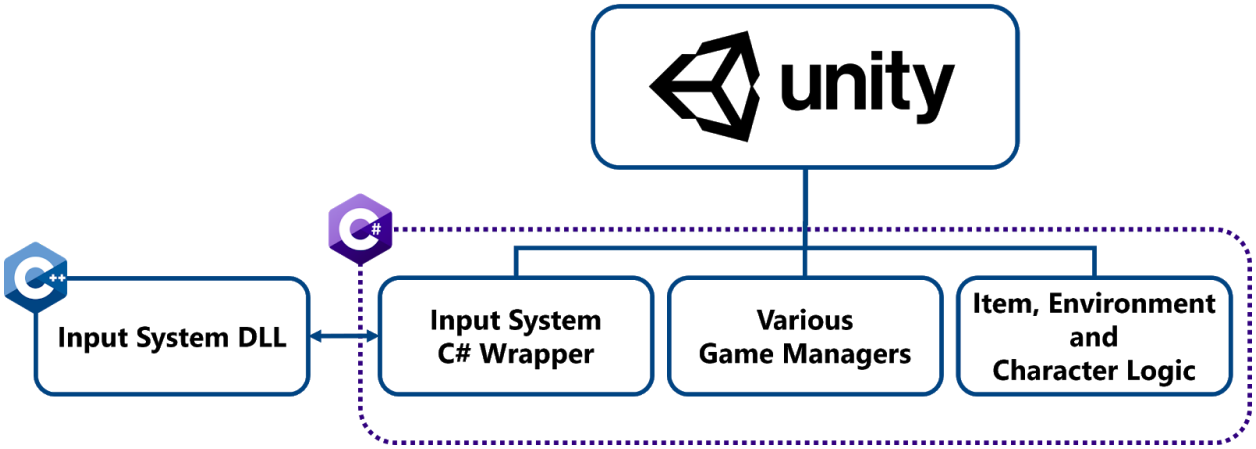

Figure 18. An overview of the architecture of *Super Collider*

Super Collider is built in Unity and uses Unity's component system and external DLLs to implement its logic. C# scripts are used for Unity components, and C++ is used for external DLLs. Unity was chosen early on in the project due to ease of use, team familiarity, and its potential for rapid iteration. *Super Collider* also makes use of Unity's built-in physics system.

Character Controller

Setup

Our character controller is based off of the work of Evan Greenwood. Our character is built using a series of rigidbodies connected by configurable joints. The X, Y, and Z motion for all joints are set to locked to prevent the joints from moving. The joints have their X, Y, and Z rotation set to limited, allowing us to specify how far each joint can rotate. The joint rotation limits are set to be similar to the normal range of motion for a human.

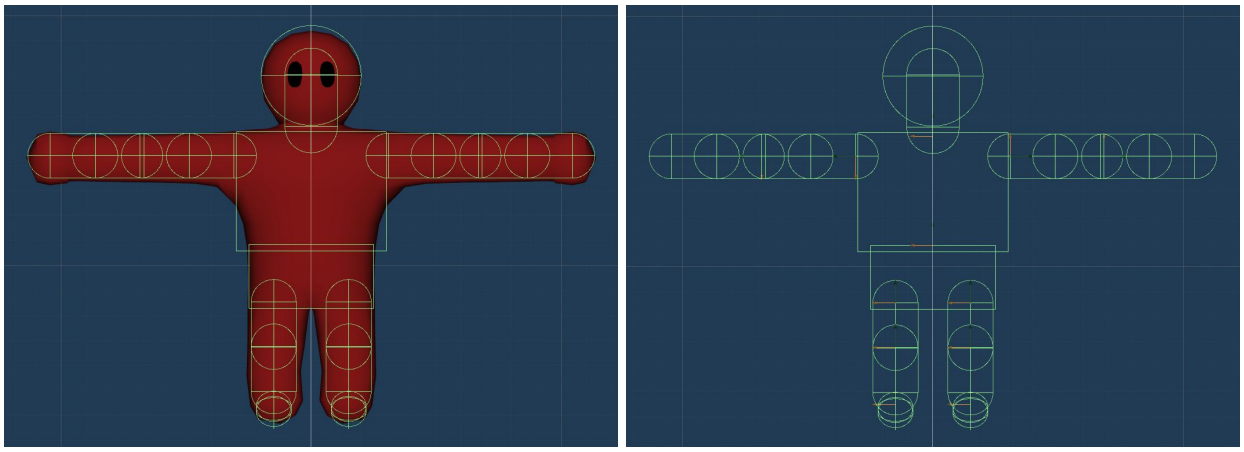

Figure 20. Primitives setup of the character controller

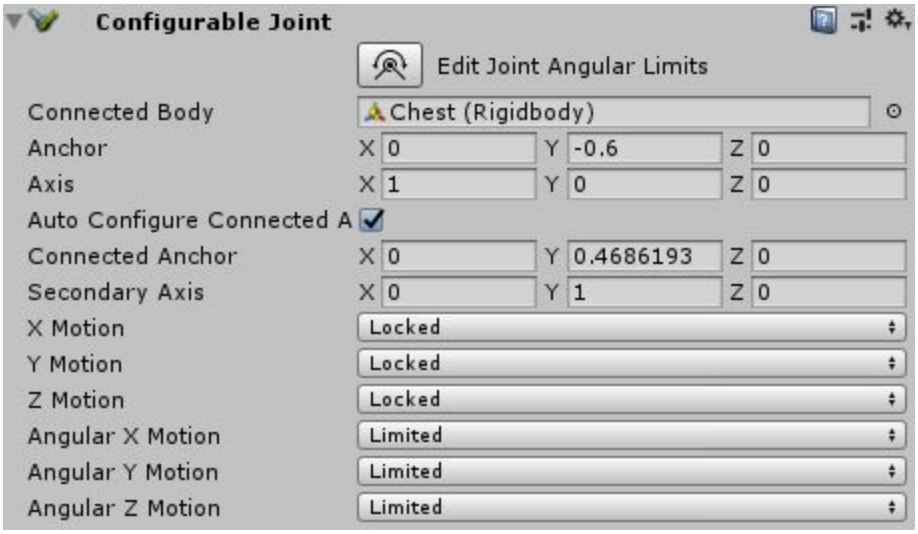

Figure 21. An example of a configurable joint

Core Components

With our ragdoll constructed, we can begin applying forces to it in order to make it move. To do this, we have three basic components that are used to define the movement of our characters.

The first component manages the height of rigidbodies relative to the ground. Ground is marked by placing objects on a specific layer in Unity. Then if ground is detected by a downwards raycast, forces are applied to the rigidbody to smoothly move it a specified distance directly above the ground that was detected. The most basic use for this is to hold up the player. While standing, the chest is kept a specified height above the ground, propping it up. In this case, counter forces are applied to the feet to keep the character from appearing floaty since this action of maintaining height will fight against gravity.

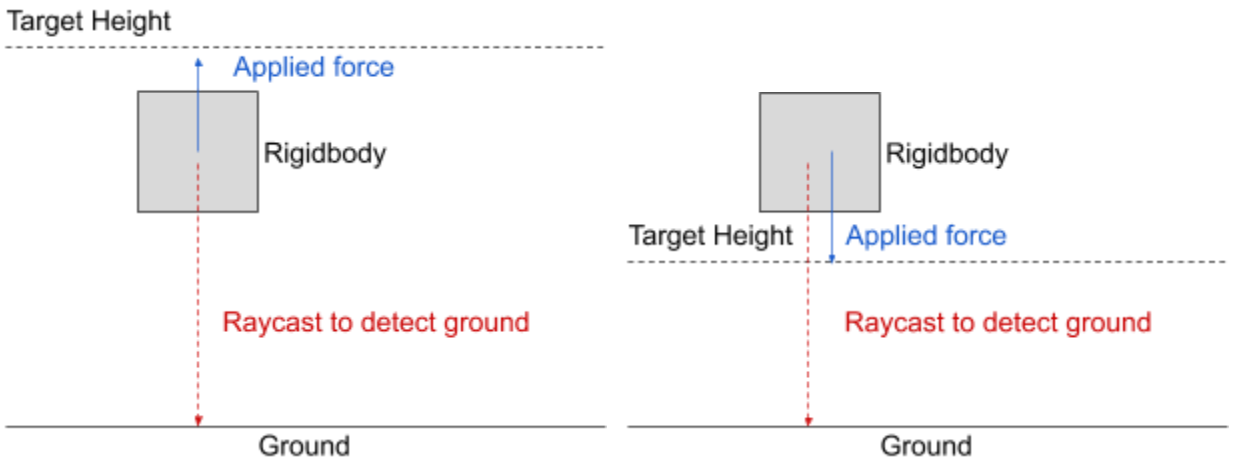

Figure 23. An illustration about how the height targeter works

The second component manages the position of rigidbodies. This component allows us to specify a position that the rigidbody should be at, relative to the transform of another rigidbody. Similarly to managing the height of a rigidbody, this component will smoothly move the rigidbody towards a target position. However, instead of using a position detected by a raycast, the position is calculated based on the transform (position and rotation) of another rigidbody. For example, we use this component to keep the character's arms to the sides of the chest. The strength of this pull is weak compared to many of the other forces that we apply to the character. This makes it so that the arms will tend towards the sides of the chest, and not snap to it, resulting in more believable behavior.

The final component manages the direction that rigidbodies should face. This is done by applying two equal but opposite forces offset by some distance along the axis defined by the direction of the forces. Think of a cube with two strings attached at the center of opposite faces. If you pull on the strings with equal but opposite forces, the faces will align with the axis that you are pulling along and the cube will not move since the net force of the system is zero. Therefore, we can make any rigidbody tend towards

facing a specified direction and scale the speed at which it reaches its target by scaling the strength of these vectors (while keeping them equal but opposite).

This component is used to adjust the facing direction of the chest. Since the chest is both the heaviest rigidbody and the rigidbody that every other rigidbody is most closely connected to, the entire system closely follows the movements of the chest. This allows us to easily make the character stand upright and turn. This component is also used on the feet to make their y-axis always align with the global y-axis, which keeps the feet from dragging along the ground. This makes walking a lot smoother. Our final model did not end up using the feet as part of the mesh, so it has no impact visually, but we chose to keep the feet since it did not impact movement and the walking logic did not need to be re-written to account for not having ankles.

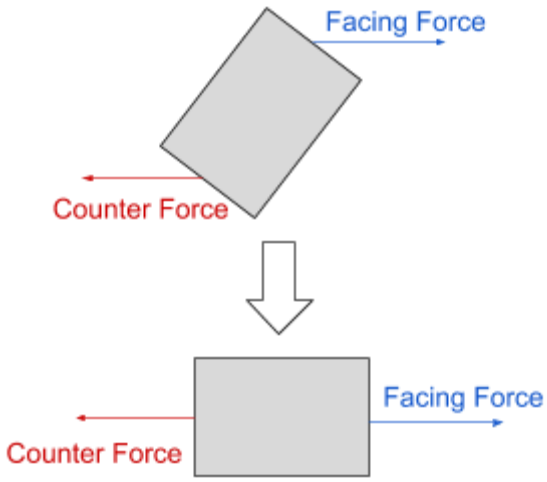

Figure 24. An illustration about how a direction targeter works

Walking

Our character controller mimics a version of the goose step for walking. The goose step is a marching step where the legs are raised high off the ground without bending the knee. This is fairly simple to implement

with our system compared to how a human would regularly walk. Performing a goose step for a single leg involves a four-step process. During this process, most forces applied will have some counter force applied to either the leg not being moved or to the chest in order to keep the character from sliding. We only want movement to come from the raising and lowering of the legs.

1. Raise the foot off the ground
    a. Apply an upwards force to the foot, raising it off of the ground. A counter force is applied to the chest to keep the character from lifting off the ground. These forces are applied through every step, ensuring that the foot remains off the ground.
    b. Begin rotating the thigh to straighten the leg out in front of the character.
    c. Apply a forward force to the foot to kick it out in front of the character as the thigh rotates. Apply a counter force to the other foot. This backwards facing counter force works the same as how a human pushes against the ground with their grounded foot to propel their body forward while walking. These forces are applied through every step, ensuring that the leg will end up in front of the character when it is lowered.
2. Raise the thigh and bend the knee - This is where our method differs from the goose step. By bending the knee while raising the foot, the foot gets off the ground sooner and prevents it from being dragged along the ground.
    a. Continue rotating the thigh to straighten the leg out in front of the character.
    b. Bend the knee by rotating the shin in the opposite direction that we are bending the thigh.
3. Straighten out the leg
    a. Switch to rotating the shin in the same direction as the thigh, straightening out the leg as the joints reach their limits of rotation.
    b. Apply a forwards force to the shin, helping the shin move in front of the thigh. Apply a counter force to the chest to prevent the character from leaning forwards.

c. We do not move on to the next step until the foot is in front of the chest. This guarantees that the character will progress forwards once the leg is lowered.

4. Place the foot on the ground
    a. Apply a downwards force to the foot, swiftly planting it back on the ground.

This process is repeated, switching between the right and left leg. Additionally, any time a foot is on the ground, a strong downwards force is applied to the grounded foot. This increases the friction between the foot and the ground, making it easier to push the character forwards without their feet sliding.

Punching

Punching is a complex action that is defined in the arm controller. When a player inputs a punch while the arms are in an idle state or during the ending lag of a punch action, a series of events will unfold. First, the HandTargetter will decide which arm to punch with using an alternating pattern. The selected arm will enter a joint state in which a light force targets joint rotations for a predefined pullback orientation. After a short delay, the player's arm will have an explosive force push it towards a target in front of the player and the player's hand will activate the component that gives the hit extra knockback and allows it to knock other players unconscious. The light forces targeting the pullback animation are left active so that the player's arm will naturally pullback after the punch. Next, the player's arms will enter a cooling-down state that can end in one of two ways. If the player does not input another punch command, their arms will go back to the idle state and the angle targeting force will be removed. However, if a player inputs another punch command while in the punch cooldown state, they will exit the cooldown period immediately and

perform another punch. When chaining punches in this way, neither arm will lose the rotation targeting forces until the arms return to the idle state. This produces a procedural animation resembling a flurry of blows with individual punching animations bleeding into each other in a consistent manner.

Picking Up Holdables

The three core components are enough to build a game similar to gang beasts, where all of the actions are done by pulling body parts (most appendages) in different directions for different actions, such as punching or controlling individual arms to climb surfaces. However, these components are not enough for all of the cases in our game. At times, we require more precise control over exactly how our appendages are positioned. Without fine control, appendages can easily become twisted up with each other. For example, if we wanted to apply a force on the left hand to push it to the right of the chest, we have no control over whether the arm will pass in front of or behind the chest as it passes from the left to right. Additionally, this method does not guarantee that connected joints such as the elbow or shoulder will always be at the same position or rotation. This is not to say that it is impossible to do with more advanced logic, but it was clear that this would require a lot of trial and error to account for all edge cases for every action we may want.

This is a problem that we had to solve for how we pick up items, or Holdables as we called them. A Holdable defines the parameters for how it should be picked up and locked to the player's hand or hands. Instead of the method described above, we can utilize configurable joints and set a target rotation for each joint. This way, for relatively simple skeletal systems, such as our characters, every joint that we set a

target rotation for is nearly guaranteed to be at the same position and rotation no matter its starting configuration. Then by using this, we do the following steps when picking up an item:

1. Move the arms into the correct position by setting the joints' target rotations to predetermined values
2. At the same time, move the Holdable into position in front of the player
    a. The Holdable will orient itself so that it faces forwards relative to the player
3. Once the arms and Holdable are within some error of their targeted positions, connect the Holdable handles to the player's hands with joints
    a. Holdables have predefined handle positions, indicating where the joints should be placed
    b. Target joint rotations and Holdable positions are made such that the hands and handles will be relatively close together when the joints are made so as to reduce snapping together from large distances

The type of joint used when joining the hands to the Holdable will either be a fixed joint or ball and socket joint. Fixed joints are used for one-handed Holdables as it mimics a person grasping an object tightly in their hands. It also prevents the Holdable from spinning around independently from the hand.

Ball and socket joints are used for two-handed Holdables because it very nicely estimates wrist movements. When holding an object with two hands and moving it in the air, a normal human's wrists are not static. They are constantly moving to help give a more comfortable hold on the object. Ball and socket joints mimic this movement without actually needing a wrist on the character, dramatically simplifying this problem, and making for a much more natural and stable hold on the Holdable as the player moves

around than if we were to use fixed joints. Additionally, since there are two joints (one hand on each handle), the Holdable is prevented from rotating.

There is an additional third case where no joint will be used to connect the Holdable to the hands. Instead, the Holdable will hover in front of the character. This is done by continuously moving the Holdable to be a set distance in front of the character. We also adjust the rotation of the Holdable to be pointing forward relative to the character's chest. Currently, this is only used for picking up players. We chose to use this approach instead of carrying them like any other Holdable because character rigs are much larger than our other Holdables and it was difficult to create a setup where the players could always get a strong hold on another player that they picked up.

Holdables

Guns

Every gun in the game operates using two connected classes, one inheriting from Holdable and another that implements the IShooter interface. This was set up in such a way that the same gun shooting logic could be used in different contexts. HGun and HStationaryGun are two different types of holdable guns that can both interface with any implementation of the IShooter interface. HGun has all of the core logic for a small, handheld gun. HStationaryGun is used for stationary turrets that the player can operate. BasicShooter and GattlingGun are both implementations of the IShooter interface and can be connected to and operated by either HGun or HStationaryGun. BasicShooter is for any gun that has limited ammo, and

GattlingGun is for any gun that uses a heating system and will overheat and explode if overused. The rail gun, the bomb gun, and the ricochet gun, all use this system.

Rail Gun

The rail gun is implemented using HGun and BasicShooter to launch a rail projectile. The rail projectile uses the basic Projectile script that activates its ExternalImpulse after a set time period, deactivates it as soon as the projectile stops moving, and destroys the projectile a set time after it stops moving.

Ricochet Gun

The ricochet gun is implemented using HGun and BasicShooter to launch an energy blast that ricochets off of walls becoming stronger with each bounce. The energy blast uses the RicochetProjectile script, which controls the projectile's speed and damage, and destroys it after a set number of bounces. The RicochetProjectile activates its ExternalImpulse after a set time period and destroys the object once it has lived longer than its maximum lifespan. The RicochetProjectile script will keep the projectile moving at a fixed speed. This fixed speed, as well as the knockback and power of the ExternalImpulse, will increase each time the projectile bounces until a maximum number of bounces is reached and the projectile is destroyed. The RicochetProjectile will also create a particle effect when it bounces and will change color to reflect its current power level.

Bomb Gun

The bomb gun is implemented using HGun and BasicShooter to launch a bomb. The bomb's behavior is defined by a Bomb component that will destroy the bomb's GameObject and spawn an explosion prefab if the object collides with a player or enough time passes. The bomb also uses an ExtraGravity component to artificially increase its bounciness. The ExtraGravity script simply applies a force to the attached rigidbody each physics step.

Rocket Hammer

The HRockHam component extends Holdable and implements the core functionality of the rocket hammer. The rocket hammer uses a simple state machine that allows it to be on or off at any time. When OnActivate is called, the rocket hammer will switch to its active mode. When the hammer is dropped or runs out of fuel, it will deactivate or destroy itself. In its active mode, the rocket hammer will expend fuel, apply a force to the head of the hammer, apply a force to keep the player's legs from flying out, override the players facing direction to help them spin, and apply a directional force based on player input. The force applied to the head of the hammer is calculated by adding three forces together. The first force is based on the way the hammer is facing and ensures that the hammer will wobble back to the correct orientation while the player is spinning. The next force is always perpendicular to the line between the hammerhead and the player's chest and ensures that the hammer will always move in the same direction that the player is spinning. The last force pushes the hammerhead away from the player's chest ensuring that the player's arms fully extend. This combination of forces was decided on through experimentation to see what would create a consistent animation. The force applied to the legs is calculated in a similar way,

applying a force to swing the legs in a circle and an inward counterforce that ensures that the legs don't fly out. All of these forces are added up and an appropriate counterforce is applied to the chest to ensure that the player spins without moving around too much. The function CharacterMovement.SetOverrideFacing is used to send a signal to the CharacterMovement component, telling it to face the character at a set angle in the direction of the spin, rather than facing them based on the current directional input. Lastly, the rocket hammer communicates directly with the input component of the player to get the direction the player is trying to move in and applies a force in that direction to the player's chest. This creates a satisfying spin animation and allows the player to control their movement while spinning.

Flail

The Flail is a very simple one-handed melee weapon. It is composed of a series of chains joined together by configurable joints that have all movement locked and rotation limited to restrict the range of rotation for each joint. At the end of the chain, attached by a similar configurable joint, is a sphere with spikes (cones) attached. The spikes have no colliders and are only for show. By spinning your character, the flail is pulled around the character by centripetal force (It was noted that many players enjoyed spinning their characters in circles, so we chose to use the action for a weapon). Upon contact with another player, given the sphere is moving at enough speed, it is able to knock back the player and ragdoll them. A flail is unable to knock down the player that is holding it.

KABOOMerang

The KABOOMerang is a single-handed weapon that can be thrown, and it explodes on contact with other objects. As its name suggests, it's a boomerang that will try to return to its original position after being thrown. Thus the difficult part is how to simulate its motion with the physics engine.

The motion of a boomerang is driven by aerodynamics. It is lifted by the force generated from different airspeed between the top and the bottom of the object, and the higher aerodynamic lift on the top end creates a torque, causing the angular momentum to precess, and gradually changing the heading.

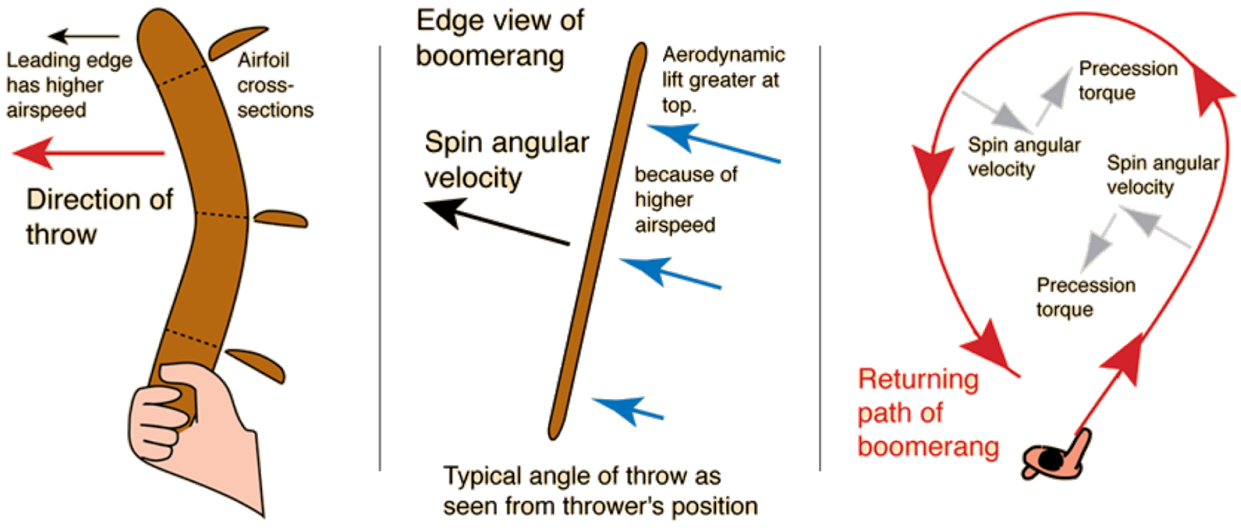

Figure 25. Illustrations about how a boomerang works

It's not practical to make a physically realistic boomerang in our game as it unnecessarily takes more time to calculate. According to [the pic above], we decided to simplify its motion into a simple harmonic motion (SHM). Though SHM doesn't precisely describe how a boomerang works, it looks good enough in our game after putting it in.

There are three key factors in a simple harmonic motion: the elastic storing force **F**, the spring constant *k*, and the displacement from the equilibrium position **x**. According to Hooke's law, the three factors obey the following equation:

$$\mathbf{F} = -k \cdot \mathbf{x}$$

In the implementation, we just need to define the equilibrium position and the spring constant to make the KABOOMerang move. When being thrown, there will be an initial velocity on the KABOOMerang, and with the initial velocity and the elastic storing force facing different directions, the KABOOMerang will eventually follow a circular or oval route, and eventually hit something (mostly the players who throw the KABOOMerang, if they didn't move).

Figure 26. The trajectory of the KABOOMerang

Cactus Grenade

The cactus grenade is a weapon that when thrown explodes into a large number of smaller cacti projectiles that cannot collide with each other. The cacti projectiles form joints with anything they collide with. The joints on the cacti projectiles use one maximum strain force for items, and another maximum strain force when they are in contact with a player. This is because the forces used to animate the player are very large and would easily rip apart the joints if they used the same force used for normal objects.

Proximity Mine

The proximity mine is a weapon that uses HProximityMine to attach to the terrain when used and explode when it detects any rigidbody entering its space. HProximityMine extends the Holdable class and implements a state machine to track whether the mine is inert, thrown, in the process of arming, armed, or has already exploded. When HProximityMine.OnThrow is called, the mine will enter its thrown state until it collides with an object on the Ground physics layer. Once it hits an object on the correct layer, it enters its "landed" state in which it will stabilize its position before activating the light on the top of the mine and entering the armed state. If the object is attached to moves, resizes, rotates, or is destroyed while the mine is attached and in the landed or armed state the mine will explode, destroying itself and spawning an explosion prefab. Additionally, if any rigidbody enters the trigger collider surrounding the mine while it is in its armed state, the mine will explode.

Plunger Gun

The plunger gun is an unorthodox weapon that allows players to shoot a plunger projectile. The main components for this weapon are HPlungerGun and Plunger. HPlungerGun inherits from holdable and contains functionality for instantiating a new plunger from a prefab and connecting it, commanding the plunger to retract, detecting if the plunger has been destroyed, and shooting the plunger. If the plunger is detected as having been destroyed, the plunger gun will simply spawn another one. The plunger is shot by detaching the joint that holds it in place and appling a large force. The Plunger component tracks if it is in an active state, uses an auto aim system when it is shot, retracts itself using context dependant logic when appropriate, applies a context dependant on collision effect, and detects if the plunger gun is destroyed.

The active state is when the plunger is not attached to the gun and is able to attach on collision. The auto aim system uses a trigger collider to detect potential targets, and uses a raycast to ensure the path is clear. The auto aim system will only aim at players or items. The plunger will apply one of a variety of retract effects based on if it is attached to an object and what it is attached to. If it is not attached to an object, it will simply pull itself back to the gun. If it is attached to the terrain it will pull the player holding the gun towards it. If it is attached to any other rigid body, it will pull that rigidbody towards the player holding the plunger gun. The on collision effect will attach the plunger to whatever it hit and produce additional context based effects. If terrain was hit, the player holding the plunger gun will be launched towards the plunger. If a weapon was hit, it will be equipped if applicable and will be launched towards the player holding the plunger gun. If it was another player hit it will launch them towards the player holding the plunger gun. If the Plunger ever detects that the HPlungerGun has been destroyed, it will destroy itself.

Rocket Glove

The Rocket Glove is a boxing glove with a rocket thruster attached to the base (where a person would insert their hand into the glove). Upon activation, the player will be pulled along by the glove and is able to control the direction that they travel. If the glove or the owner of the glove's head collides with another player at a fast enough speed, the other player will be knocked back and ragdoll. The rocket glove can be controlled for up to a few seconds before it is automatically deactivated, but the player may choose to deactivate it early.

The main problem to solve with the Rocket Glove was how to best get the player into the air while it is being used. This serves two purposes. Firstly, it serves as a visual element, as it is very fun to watch the character get pulled through the air like a cartoon character. Secondly, it serves a physical element, as lifting the player off the ground removes friction, making the movement of the glove and player much smoother. This is achieved by adding a component that moves a rigidbody to a target height (explained in the Character Controller section above) to the glove and the character's chest and feet. The strength of these components are adjusted to just barely hold the player up, making it look like they are floating instead of snapping to the target height. Then, a forward force is applied to the glove in the direction the player wants to move and an opposite but weaker force is applied to the feet to keep the player behind the glove. Keeping them behind the glove further sells the feeling of being pulled by the gloved and prevents the character from becoming tangled with itself.

While activated, the Rocket Glove has a fire trail that is shot out behind it. This is done by using a series of particle systems that are set to play while the Rocket Glove is active and set to stop when it is deactivated. It should be noted that this is different than just enabling and disabling the particle systems since doing so would remove any alive particles. Stopping the particle system will allow the alive particles to continue through their lifetime.

Laser Cannon

The Laser Cannon is a powerful weapon that involves charging a laser with energy until the energy is released at once in a single, powerful burst. If the charge is released before the burst, the gun will not fire

and will need to be fully charged again. While charging, we use a gradually growing sphere at its muzzle to indicate how charged the gun is and spherical particles are pulled into the sphere, helping indicate to players that the fun is charging. The burst is the same dissolving laser used in the spinning laser level and the lasers seen on the conveyor level (How the lasers work will be explained later). The width of the laser as it is firing is controlled by an animation curve, quickly bringing the laser up to its full width and then smoothly fading it away.

The Laser Cannon has a grill on it that displays some blooming light and spins around. This grill adds some visual flare to the gun. As the gun is charging, the speed that the grill spins will gradually increase and after it fires or is released early from charging, it will gradually return to its original speed.

Input System

There are several reasons why we need a custom input system. Prior to Unity 2019.1, the built-in input system is not well-designed for a multiplayer game. Things like the same controller can be assigned a different index after reconnection always happens, which is not acceptable in a multiplayer game. Also, the built-in input system doesn't support controller-specific features like vibration and LED light bars.

Super Collider is only targeting Windows currently, thus we decided to have our input system interfacing with Windows native input API XInput directly. We also supported DualShock 4 controllers of

PlayStation 4 with the Windows HID API. But in case the game releases on other platforms, we want our input system to be extendable.

Above that, we also want to have a flexible key/button mapping that can map the actions in the game to actual buttons on controllers, and we can modify the mapping in the inspector or in-game with scripts.

And finally, we want our input system to have a clear and easy-to-understand API. Figure 27 describes the design of our input system.

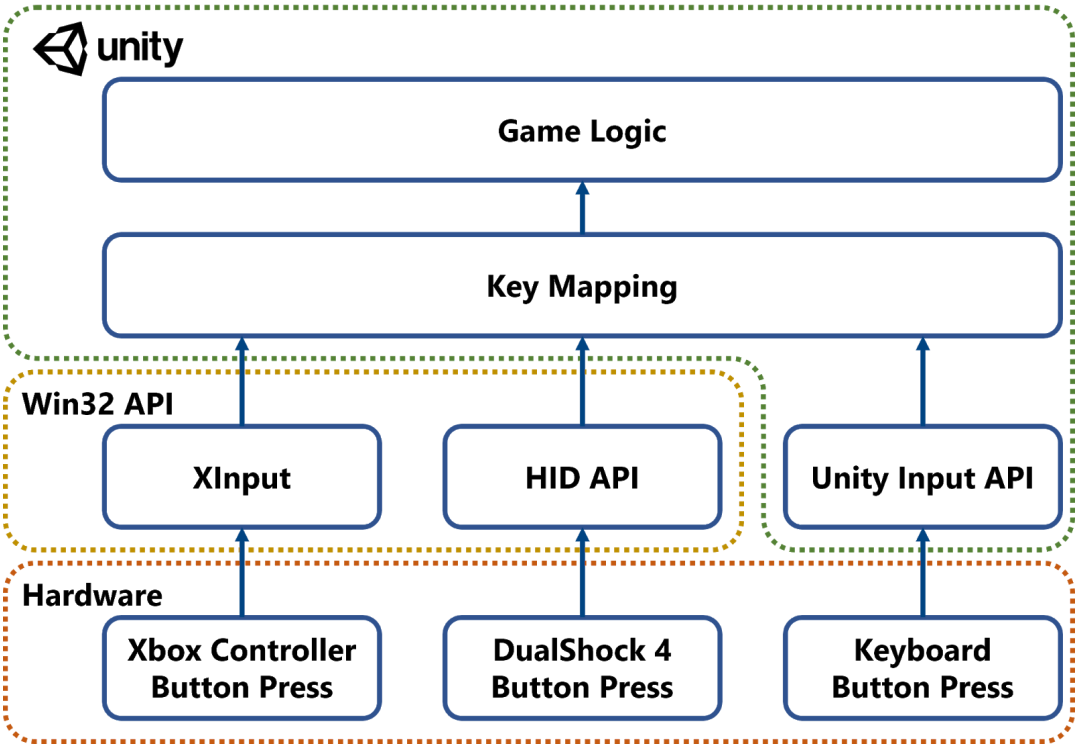

Figure 27. An overview of the architecture of the Input System

The first thing to do is to have the Unity project be able to call functions from XInput. We have a DLL written in C++ and the functions from it are imported into C# scripts.

The C++ wrappers, by design, should have minimal functionalities. It should only handle things like getting the value of an axis or getting a button press action. In the implementation, the wrapper API we expose to C# scripts include things like initializing XInput, checking the controller connection status, updating the button status of the controllers, and getting a button press/release/hold action.

XInput only supports Xbox 360 controllers and Xbox One controllers, but we also want to support DualShock 4 controllers from another well-known console PlayStation 4. The DualShock 4 controllers can be connected to a Windows PC through a USB cable or Bluetooth and will be recognized as a DirectInput device. However, DirectInput API is deprecated and not recommended by Microsoft officially, so we decided to use the human interface device (HID) API to talk with the DualShock 4 directly. This will also allow us to use some DualShock 4 specific features like the touchpad and the LED light bar.

Instead of using a C++ DLL like we did for XInput, we directly call the C++ Win32 API from C#. This allows us to dynamically identify the Windows version at runtime and decide if the API going to be called

is available. These Win32 API will handle things including listing all HID devices, getting the vendor ID and the product ID of an HID device, and reading/writing packets from/to a device.

To communicate with a DualShock 4 controller, we first need to know its specifications. Fortunately, a lot of people have already done that. The vendor ID and the product ID we use to identify if a device is a DualShock 4 can be found in the Steam controller database. We also found the data packet definition from PS4 Developer wiki. With all that data, we are able to identify a DualShock 4 controller from all HID devices and utilize all of its features, including the LED light bar.

With the ability to communicate with different kinds of controllers, we need to have a way to map the inputs to actual actions in-game. We also want this mapping to be easily modified in C# script since we originally would allow players to change the key mapping in the settings. We used Unity scriptable objects to describe such a mapping relationship.

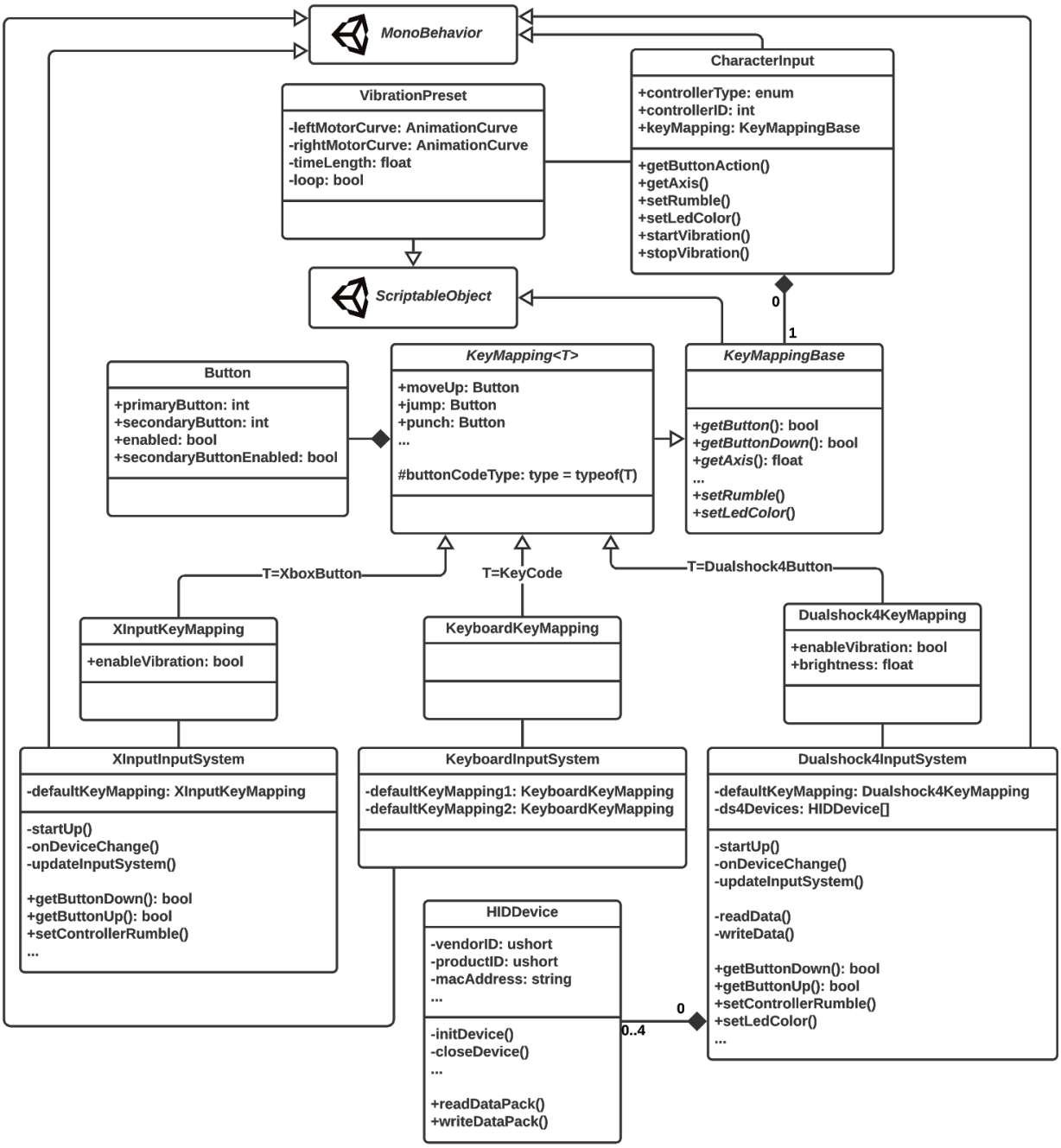

Figure 28. A class diagram for the Input System

We first defined some basic common things like the in-game actions (jump, punch, etc.) and abstract methods (get button down, set LED color, etc.) in the KeyMappingBase class, then we derived it with a generic class KeyMapping<TButtonCodeType>, where TButtonCodeType is an enum containing all of the buttons supported by the controller. Inside the generic class, all actions are defined so that we can use it to change buttons in the inspector. The actions are defined as a Button type, where Button is a class holding the info of the primary button and the secondary button. Both of these info are integer types because we need Unity to serialize the Button class so it can't be generic, but it's easy to cast between integers and enums. We wrote our own editor script to display the integers as enums in the inspector using the C# reflection. Inside this generic class, we also defined public methods for the in-game input scripts to call them and abstract methods that differ from controller to controller.

Then we need to instantiate the generic class because Unity doesn't know how to serialize a generic class. There are three kinds of controllers supported in *Super Collider*: keyboard, Xbox controllers, and DualShock 4 controllers. There is one key mapping type for each of them, and it defines the enum of all buttons on the controller, and it also implements the abstract methods so that each class can use a different approach to get the button action with a unified API.

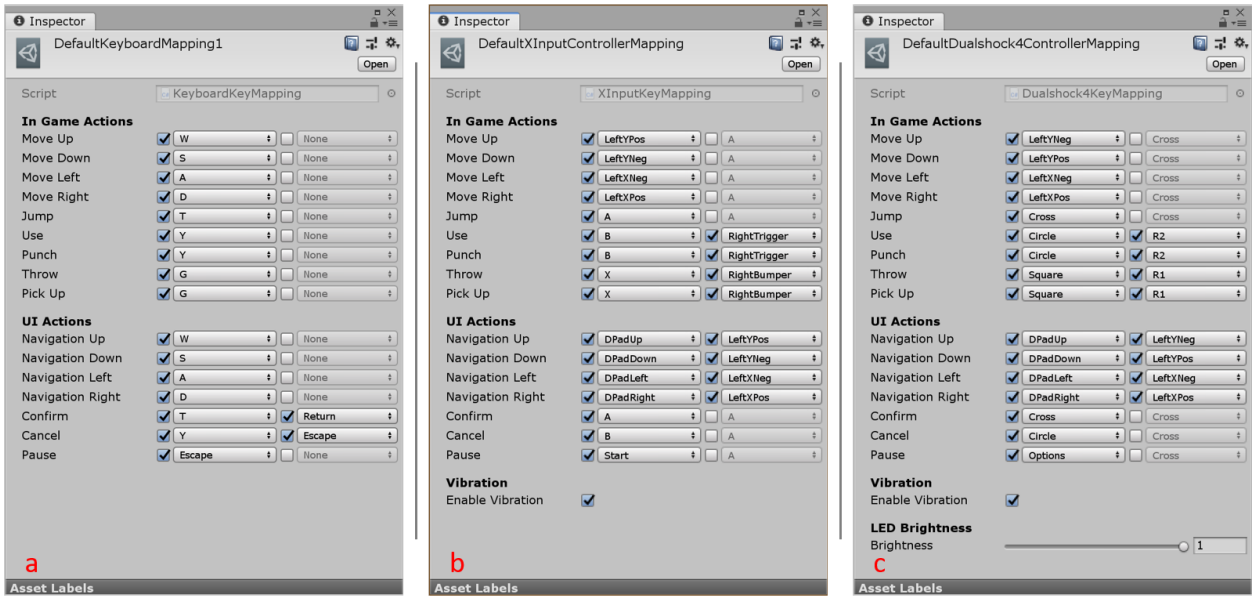

Figure 29. Screenshots of a keyboard mapping (a), an XInput mapping (b) and a DualShock4 mapping (c)

Finally, we used a MonoBehavior script called CharacterInput to hold the key mapping and the controller ID for each real-world player. The CharacterInput script defines some methods that will be used for other scripts, and these methods call the methods in the key mapping scriptable object.

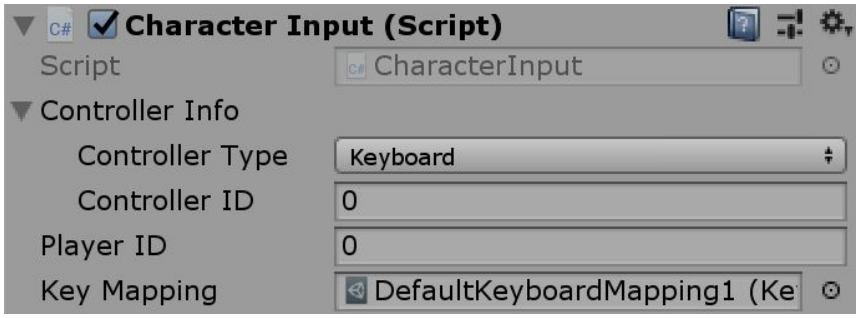

Figure 30. A screenshot of a Character Input component

After we finished our own input system, we looked into the new input system provided in Unity 2019.1 and later, and found out that we follow the same design: a mapping asset file, an input script that holds the mapping asset, and an input system running in the background. So it is safe to say that the design of our input system has already been proved better than the original Unity input system.

For vibration, we used another scriptable object to describe different patterns. The strength of the two rumble motors are defined as animation curves, and these curves are evaluated and applied to the hardware inside a coroutine and can be easily tested without putting it into the game. This makes it a lot easier for us to design controller vibrations.

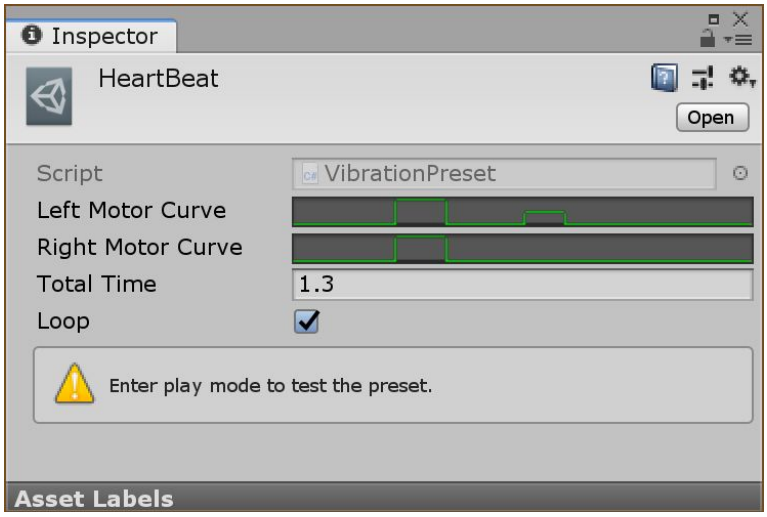

Figure 31. A screenshot of a Vibration Preset

AI Controller

Though there are no AI-controlled players in our game currently, we do have a system that supports the AI system under the hood. The AI controller was designed for the tutorial that explains the controls in-game in a visualized manner, but it is also capable of situations that are much more complex.

As our game is heavily based on physics, we can't just let the character play some animation and move its position directly. To make the AI character look like a common player-controlled character, the best way is to make a specialized CharacterInput component, which is called AICharacterInput.

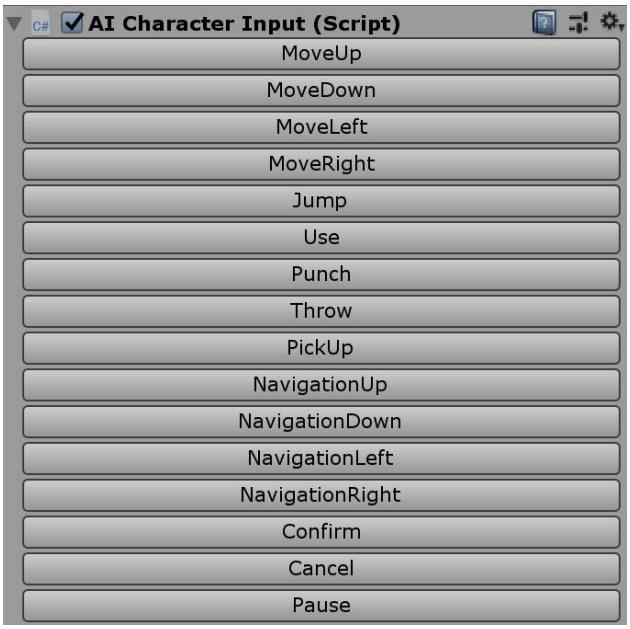

Figure 32. A screenshot of an AI Character Input component

Unlike a regular CharacterInput component, the AICharacterInput component doesn't have a key mapping asset or controller info. While it provides the same API as the CharacterInput does, it also provides APIs letting other scripts simulate a button press/hold/release action, and some helper functions like press a certain button for a period of time. It doesn't require an input system to work since all of the button state updates are carried out inside the script itself. We wrote an editor script so that we can control an AI character directly from the inspector, like Figure 32 shows.

Menu

The technical design goal of the menu UI system is that it can fit all use cases in our game with a common codebase. There are three menus in our game: the main menu, the pause menu, and the settings menu. The codebase of the menus is made up of two parts: a Menu class, and a MenuButton class. The MenuButton class defines four abstract callback methods: OnSelected, OnDeselected, OnPressed, and OnReleased. It also defines four abstract coroutines accordingly for playing the UI animation. The Menu class holds a list of MenuButtons and defines the methods and coroutines for calling the callback methods and animation coroutines. With these basic settings, we can easily extend the functionality for both the menu itself and the buttons of the menu.

Main Menu

The main menu is a kind of one-directional menu, which means that the buttons are aligned in a line and only two buttons are used to navigate in the menu, either left and right, or up and down. Instead of just UI

elements, the main menu and its buttons are 3D objects in the game world. When navigating through the menu, the camera will move to focus on different objects, like the computer for the Settings button, or the exit door for the Exit button.

Figure 33. A screenshot of the main menu

The virtual camera system, as a part of the main menu, was designed and implemented during the time we decided not to use the Unity Cinemachine. It is made up of two parts: several virtual cameras, and a virtual camera controller attached to the actual camera. The VirtualCamera classes hold a static list of all VirtualCamera instances currently in the scene. When the priority of a VirtualCamera is changed or a VirtualCamera gets enabled/disabled/created/destroyed, a static method will be called to update and sort

the list based on the priorities of all instances. When updating the list changes the VirtualCamera with the highest priority, the transition coroutine of the VirtualCameraController attached to the camera will be called, moving the camera to the desired position and rotation.

Pause Menu

The pause menu is also a kind of one-directional menu. There are only two buttons in the pause menu: a resume button and an exit button. The player can unpause the game by pressing the resume button or the B button on the controller. The pausing/unpausing is implemented by changing the time scale, thus all time-related things in the pause menu, and eventually the one-directional menu should use the unscaled time to prevent the pause menu itself being paused.

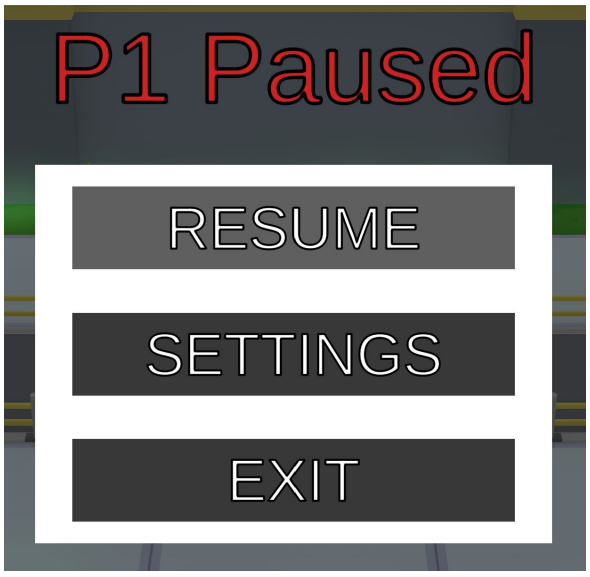

Figure 34. A screenshot of the pause menu

Pressing the exit button will seemingly restart the game from the main menu, but there are things going on under the hood. First, it will destroy most of the objects that are tagged with DontDestroyOnLoad since the game is meant to restart, but things like the audio manager, the settings manager, and the coroutine proxy that runs coroutines across scenes should be remained not destroyed. And it will also restore the time scale to 1 since it is called from the pause menu.

Settings Menu

The settings menu is made up of two parts: a tab and a panel, which contains several settings entries. Both the tab and the panels are one-directional menus.

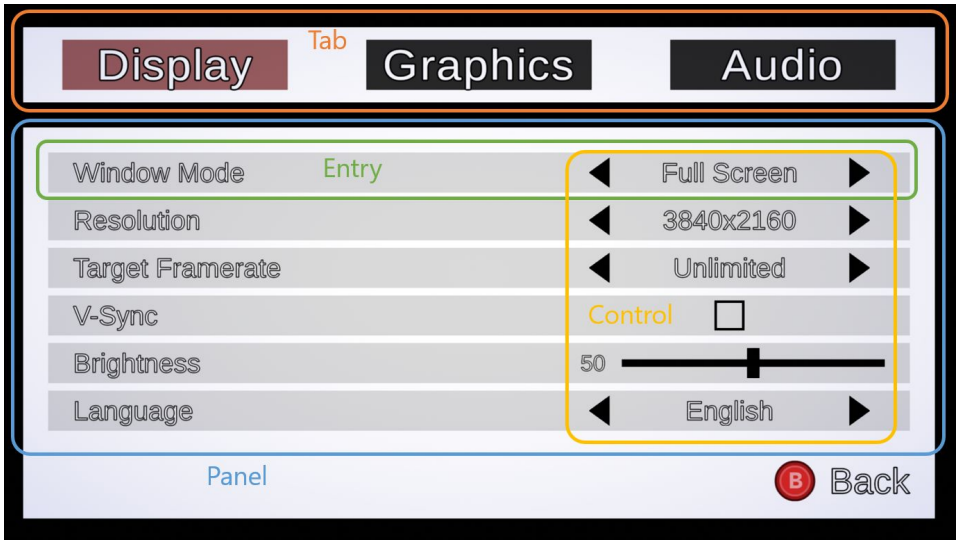

Figure 35. A screenshot of the settings menu with details

By selecting a different tab, a panel will be shown accordingly. Pressing the tab will enable the panel respectively, and allow the player to choose the entry to change.

There are three kinds of settings in the game: Display Settings, Graphics Settings, and Audio Settings. Applying these settings will save the data to both the settings manager and a config file. The settings manager will read the config file and set the settings at the beginning of the game.

The Display Settings include things like window mode, resolution, V-Sync, etc. After being applied, the settings manager will change the Unity quality/application/screen settings accordingly. The Graphics Settings include anti-aliasing mode, SSAO, and bloom. These settings will be applied and pulled by a script attached to the main camera setting the post-processing stack/volume. The Audio Settings include master volume, music volume, and SFX volume. These settings are automatically applied to the settings manager and the audio manager on changes.

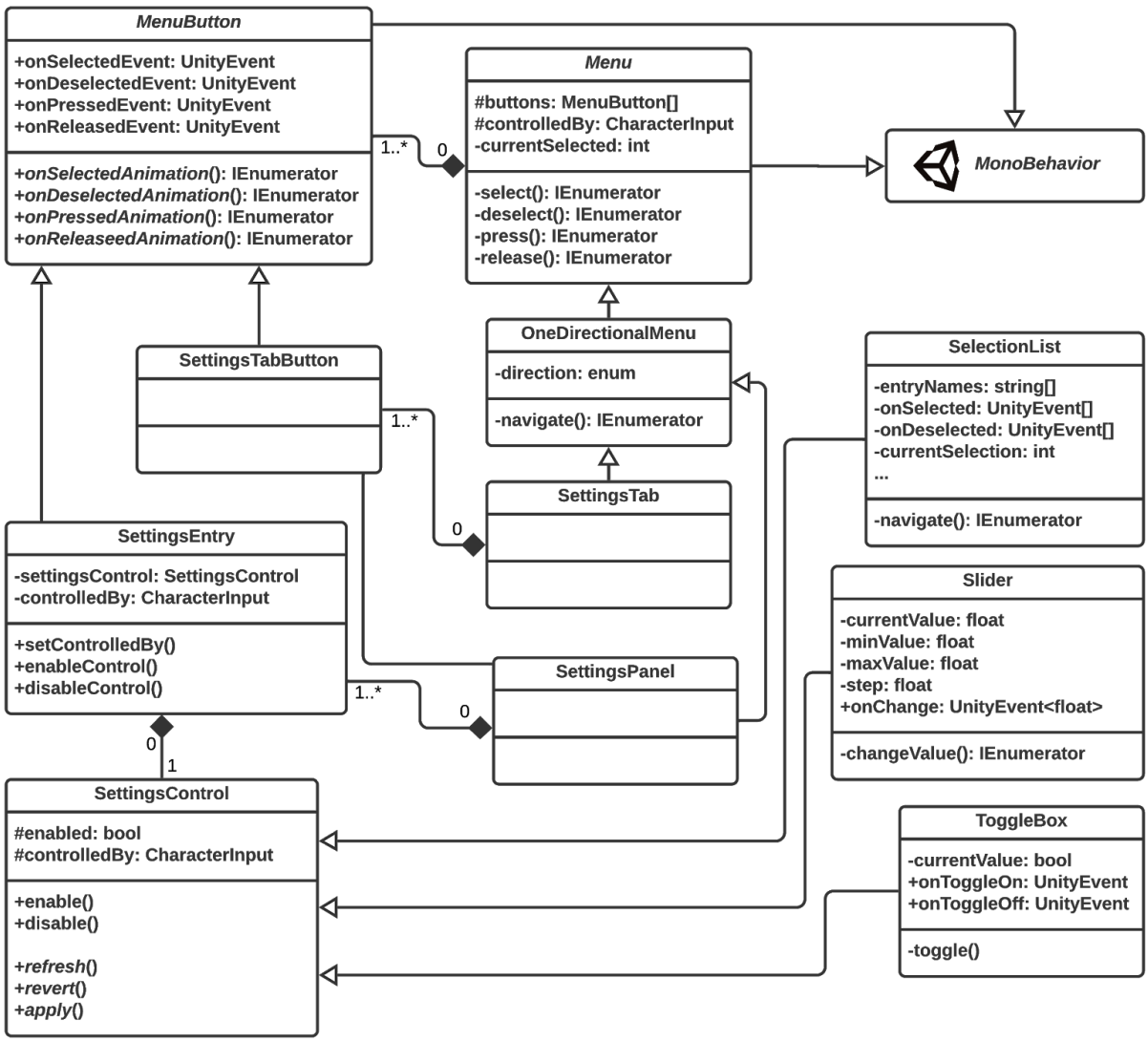

Figure 36. A class diagram for the Settings Menu System

There are several kinds of controls: selection list for enum values like window mode, slider for float values like audio volume, and toggle box for boolean values like V-Sync. All these controls are inherited from the SettingsControl class. A SettingsControl is a composition of a SettingsEntry, which is a MenuButton that can't be pressed. When a SettingsEntry is selected, it will activate its SettingsControl,

allowing the player to change the settings. SettingsEntry is a composition of SettingsPanel, which is a kind of OneDirectionalMenu, and the position of a SettingsPanel is controlled by a SettingsTabButton. When a SettingsTabButton is selected, the SettingsPanels will be moved to the desired position accordingly. When it is pressed, the SettingsTab, which is an OneDirectionalMenu, will be disabled, and the respective SettingsPanel will be activated.

Lobby

For a multiplayer game, the lobby is one of the most important parts. It assigns different controllers to different players and provides individual settings for each player. The design of the lobby in our game has gone through several iterations.

The first version of the lobby serves as both a lobby and a simple tutorial. At the very beginning, there's nothing but a flat button saying "Start Game" laying on the ground. When a player pressed the A button, a character with a predefined color will be dropped from the sky, along with a huge controller model (shown as a cube in Figure 37). Either standing on the Start Game button or placing the controller on the button will set the player's status to Ready, and after all of the players are ready the game will start after a 3 seconds count-down. The players can also move around the space freely and punch each other, which will give the players a brief introduction to the feeling of the game.

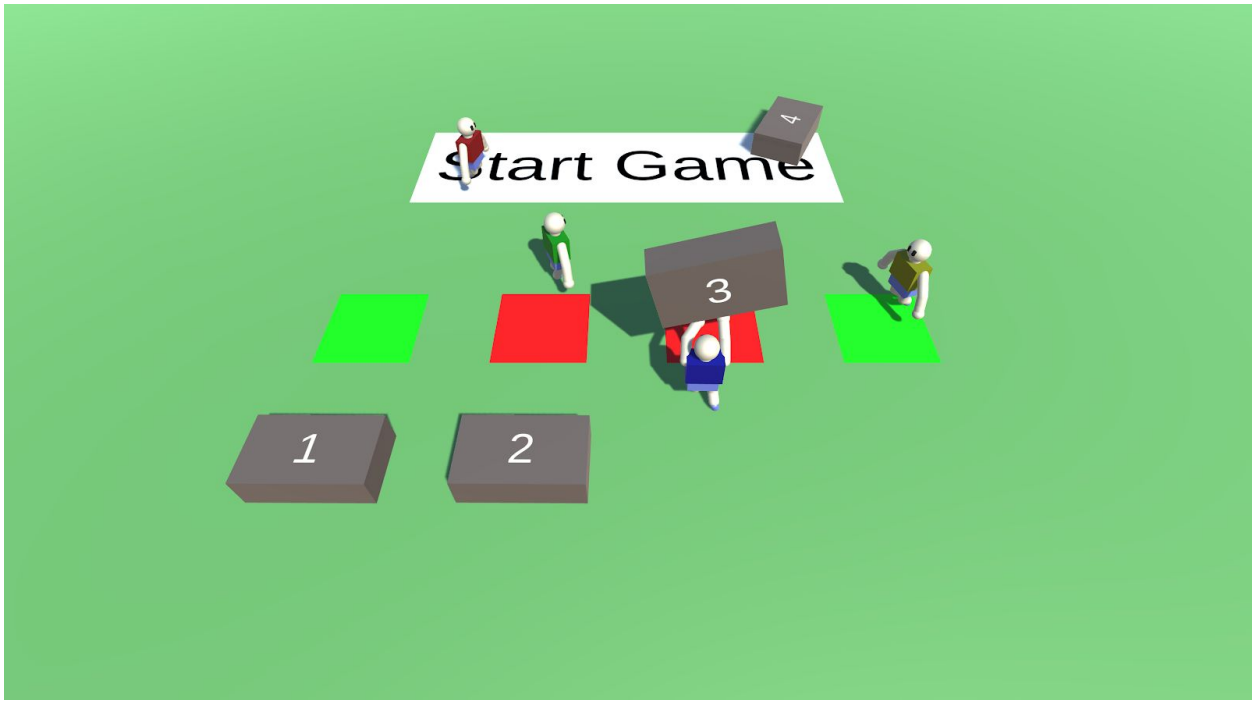

Figure 37. The first version of the lobby

The first version was designed around the time when picking up objects only requires the player to walk to the object. So the controllers are the first objects that the players can pick up during the game. This design also allows players to pick up others' controllers and force them to ready if they are not cooperating.

The second version removed the controller models. There are two major reasons for that. The first one is that we changed how the characters pick up things in-game, thus we don't need to have this kind of intuitive tutorial; the other one is that players tend to get confused when they see the controllers and don't

know that the controllers also serve as a way to get ready. In this second version, we also added a text indicating that the player should stand on the Start Game button to get ready.

Figure 38. The second version of the lobby

The third version, which is also the latest version of the lobby looks more like a common one in most multiplayer games. We totally removed the ability of the players being able to move around and punch, and all they need to do to get ready is to press A button. Instead, we let the players choose their colors on their own instead of providing a predefined color set. This is based on the fact that the players always forget their colors during playtests, and we think that letting them choose colors will help reduce the chance for them to forget their colors.

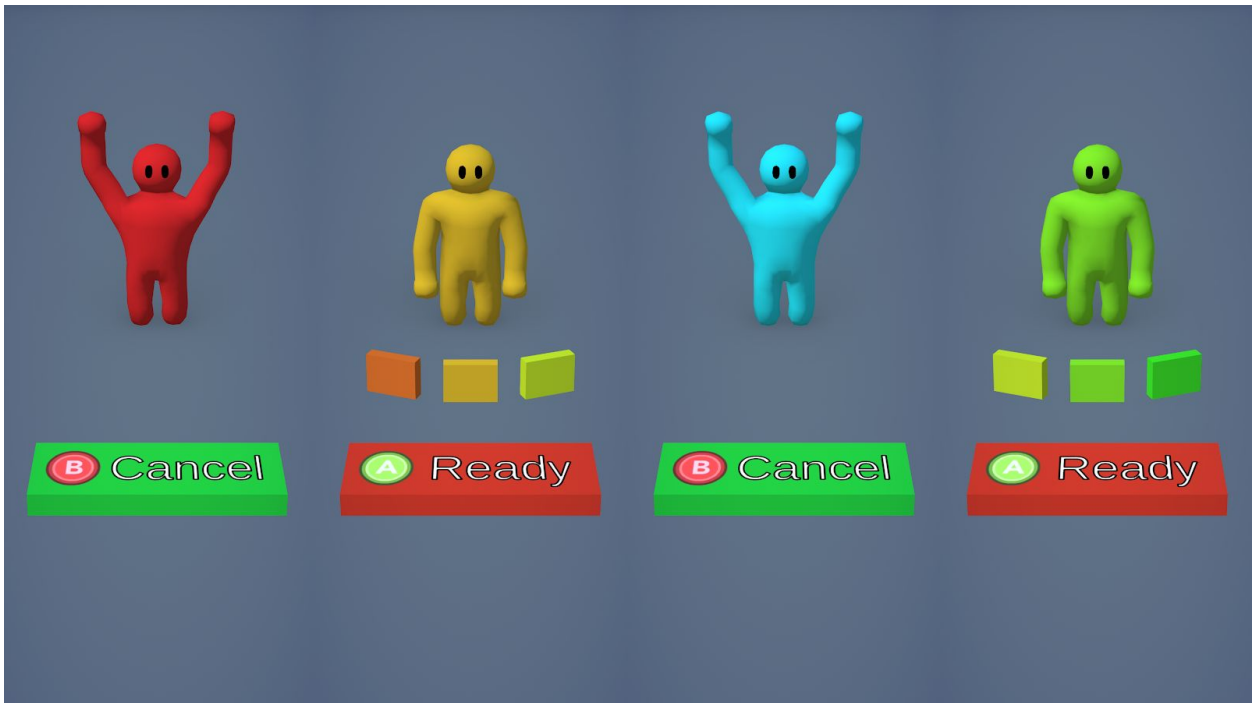

Figure 39. The final version of the lobby

Game Manager

The game manager manages everything needed to play through a full game - loading and moving between levels, spawning players, determining winners, and keeping track of score. This is managed through a series of coroutines that act as the main game loop. Each coroutine represents a different part of the main game loop:

1. Load the next level - Each level is separated by its own scene. We need to be able to load the scene and then place it to the right of the previous level before we transition to it. To do this, there

is a single root object in each level's scene. We load the scene and then move the root object to the position of the next level, then detach all of the children from the root object and delete the root object. We detach the children because a lot of our scripts rely on specific objects being roots.

2. Transition to the next level - The camera zooms out a bit, pans to the right, then zooms back in on the next stage. The previous level is then unloaded. Since the previous level is in its own scene, we only need to unload the scene to clean up anything left over from it.
3. Spawn the players - The players are spawned and brought in on drones.
4. Start the round - Once the drones are all in position, they simultaneously drop the players. This triggers an event that notifies a level has begun. Different objects may be subscribed to this event to begin their tasks, such as turning on conveyor belts, starting up dispensers, etc.
5. Play the round. Play continues until one player remains.
6. End of round - Award the winner with a point, display the score, and prepare to start from step one again. If the target score has been reached by a player, load the results screen instead.

Audio Manager

The concept of an audio manager is very simple, it should be able to play any music or SFX clip that you request. Additionally, we wanted our audio manager to manage Unity's Audio Sources, adjusting their priority, volume, and pitch as needed.

By default, Unity can play 32 sound clips concurrently. This number can be adjusted, but at the end of the day, Unity can only output so many sounds at the same time. However, you can still play more than that max number, even if you won't hear all of them. Unity will prioritize sounds, first by the priority set on the Audio Source that the clip is playing from. Higher priority sounds (indicated by a lower priority value) will be selected before lower priority sounds. If there are still too many sounds playing after culling lower priority sounds, Unity will do an additional pass, culling sounds based on their loudness - louder sounds will be chosen over softer sounds.

When playing a sound, the Audio Source will be assigned a priority based on the sound being played. Thus, we know that specific sounds will always be heard even if there are too many sounds concurrently. We still want to be careful with how many sounds we're playing, since we don't actually want sounds to be cut out, but this is a nice safety net to ensure that the sounds most important to the players are always heard.

Like all managers in our game, the Audio Manager is a singleton. We first check if an Audio Manager exists. If one does not, we set a static variable that can be accessed globally by any other script that needs to use it. If an Audio Manager already exists, we destroy the duplicate. This is useful because it allows us to put our managers in multiple scenes and not worry about multiple instances of the same manager interfering with each other.

We use a pool of AudioData objects to hold the information about all of our sounds (which is just a simple class that holds an AudioSource and a volume value as a float number). We create a number of sources greater than the number of sounds that be concurrently played (32 in our case) for the reasons discussed above - Unity will handle which sounds are actually heard based on their priority. The function used to play sounds is a coroutine. At the beginning of the coroutine we remove an Audio Source from our pool, then we play the sound, and once the sound has finished we place the Audio Source back into the pool.

As with any game, we have the ability to adjust the volume. Our game specifically supports being able to adjust the volume of all sounds (master volume) as well as the volume individually for the music and any sound effects. This is why we need to store the original volume of the clip being played - it preserves the original volume while allowing us to apply multipliers to the value based on the volume settings.

Also, with each sound played we have the option to very slightly vary its pitch and volume. This makes it so that no two instances of the same clip will sound exactly the same. This is a lot more interesting for players since they won't have to listen to the exact same sound every time. This is especially useful for sounds that are repeated in quick succession. It's also helpful for a small team such as ourselves because it means that we don't have to produce a bunch of the same sound that sounds slightly different than each other - we only need to make one version. This is also why we need to have a separate source for each sound played. PlayOneShot can be used to play multiple sounds on the same source concurrently and give each sound its own volume, so you could theoretically just have an AudioSource for each priority and still

be able to adjust the volume of the sound. However, the pitch cannot be controlled by PlayOneShot, it can only be modified per AudioSource.

Tutorial

Instead of instructions on how to play the game with only texts and images, we decided to have a visualized tutorial in-game walking the player through all the actions. This makes the tutorial a special level.

The tutorial level progresses after all the players perform the required action. There are walls that prevent the player from dying around the arena, and after all the required actions have been performed, the walls will be lowered, allowing the player to kill each other, thus turning the tutorial level to a common level.

Figure 40. A screenshot of the tutorial level

The technical design of the tutorial level should be non-intrusive. That means we shouldn't inject any code into common scripts that are not related to the tutorial. Those scripts include all character-related scripts and all weapon-related scripts. Thus, to manage the progress tutorial, we use a script called TutorialProgressChecker and several other scripts to be attached to the players or the weapons according to the stage of the tutorial.

The TutorialProgressChecker is the manager of the tutorial. It is a very simple state machine that goes through different stages of the tutorial. Once the requirement of a stage is fulfilled, it will go to the next stage. To limit the usage of huge switch-case statements in the update function, there are two arrays in the script: a delegate function array and a coroutine array. When the state changes, the only thing to do is to increase the number of the current state, then in the update function we just need to call the right function or start the right coroutine from the arrays. The update functions are used to check if the players finished the required action if those actions can be directly detected from an outside script, and the coroutines are used for the transitions between stages, including animations, attaching/detaching detector scripts to players/weapons, enable/disable certain controls of the players, etc.

There are several kinds of detectors that should be attached to the character objects accordingly. For easy actions like jumping, as soon as the players pressed the right button, they can be tagged as finished. The punching stage requires the player to actually knock someone in the stage down, so the punch detector will check if the player/dummy being hit by the current player enters the ragdoll state after the player pressed the punch button. The use item detector and throw item detector is basically the same as the jump

detector, but it only starts working after the player picks up an item. Stages like picking up items can be directly detected from the progress checker by detecting the state of the hands, so there is no dedicated detector for that.

There is also a hologram character inside the tutorial level. It will keep performing the action of the current stage for demonstration. Resizing the character will cause some physics problems, so to make it look bigger, we put it closer to the camera to create an illusion. The hologram character is driven by the AI controller.

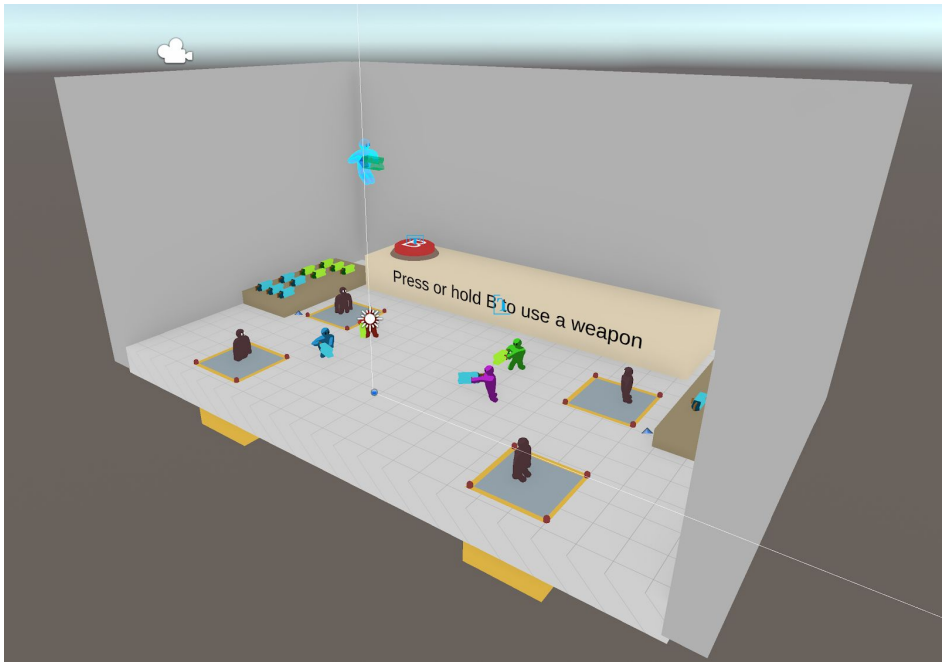

Figure 41. A screenshot of the scene view of the tutorial level

In-Game UI

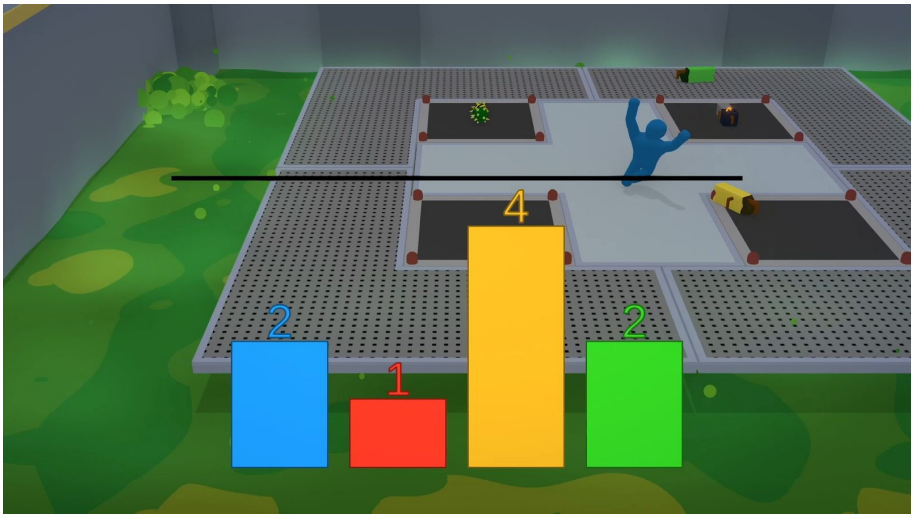

Figure 42. A screenshot of the scoring UI

The UI in *Super Collider* uses the model-view-controller design pattern. All of the data required by the UI is stored in the game manager or on the players. The data can be read by the UI through public accessors. This approach keeps the UI decoupled from the rest of the game, meaning if we change the design of the UI (we want to add/remove data that it displays) then we do not need to modify how the game functions. The UI merely operates on top of the game.

The score UI that appears at the end of each round is a canvas set to the Screen Space - Camera rendering mode. This allows the camera to exist in the 3d world while retaining its size and position as if it were overlaid on top of the screen (as normal UI canvases are). Doing so, allows us to use components only

available in world space (not UI space) such as particle systems. The score UI stores the scores from the previous round and updates its view to the scores from the concluding round through a short animation.

The prompt to mash A to get up when a character is knocked down is a series of images on a canvas. Each player has their own canvas for displaying UI elements. Splitting UI up into multiple canvases reduces the number of times the UI needs to be recalculated, so the multiple canvases are used where it makes sense. The mash A prompt is set to track the player's chest so that it is always on top of the player. This is done by converting the chest's world position to screen space position and using this screen space position as the position of the prompt.

Custom Shaders

Acid

The acid pool shader is based on a short tutorial by the Couch Game Crafters. The vertex shader generates a height offset using a cascading sine wave function with the world position of the vertex and offsets the y position of the vertex by this amount. This creates the subtle wave movements for the pool. Then, in the pixel shader we generate the height offset with the same cascading sine wave function, but with the world position of the fragment being drawn. For a little extra flare, we offset the world position used by sampling from a distortion map. We then pick a color that is a mix between two colors based on the value - 0 is dark green and 1 is light green. This gives us a smooth transition from dark to light as the height

offset moves from 0 to 1. We then take these smooth color transitions and break them up into segments using a series of step functions, giving us the banded look in the acid pools.

Portal

The portal shader is based on a short tutorial by the Couch Game Crafters. In the pixel shader, we begin by converting the uv coordinate (which is in cartesian coordinates) into polar coordinates. In the conversion to polar coordinates, we treat ‘u’ as the distance from the center and ‘v’ as the angle. Additionally, we apply an additional amount to the angle as we move outwards from the center, which will make the texture rotate faster as the distance moves outwards. We then sample a generic perlin noise texture using this polar uv value. As we offset the x position of the texture, it will look like the texture is being sucked into the center of the portal. As we offset the y position of the texture, it will rotate the texture around the center. Finally, we do the same final step from the acid pool shader, picking a color between two set colors (light and dark purple) then creating bands of color by applying a step function to the value.

Item Spawning

Ground Dispensers

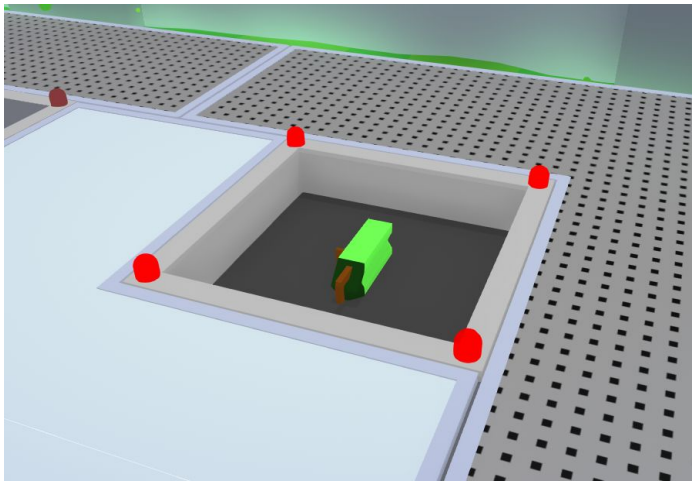

Figure 43. A screenshot of the ground dispenser

Ground dispensers were designed as an elevator that delivers items from beneath the level. When delivering items into the level, the ground dispenser follows the following process:

1. Blink the lights on the corners of the dispenser. This warns the players that items will be spawning there soon. This is done through a simple color change on a fixed interval.
2. Spawn the items in the dispenser. If there is only one item, the item is placed in the center of the elevator. If there are multiple items spawned at once, the items are placed in a circle around the center of the elevator. The distance from the center that multiple items are spawned can be controlled, allowing for different sized elevators.

3. Open the hatch and raise the elevator. This delivers the items into the level. This is done through a simple lerp function on both the hatch and elevator.
4. Wait some time. The elevator will sit in the level for a number of seconds.
5. Lower the elevator and close the hatch. Once the hatch is fully closed, all holdables and characters inside the dispenser are destroyed.

This process repeats at variable intervals. Game designers can adjust the minimum and maximum time between when the dispenser begins this process. The counter for this timing does not tick while the dispenser is active. Each step is a separate coroutine that waits for the previous coroutine to finish. By implementing it this way, each step can be individually modified without having to worry about when the other steps need to start or stop.

Wall Dispensers

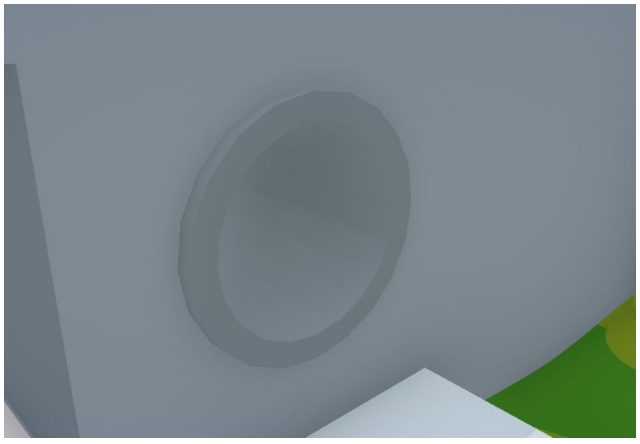

Figure 44. A screenshot of the wall dispenser

Wall Dispensers work similarly to ground dispensers, but without needing the elevator that raises and lowers items. Instead, the dispenser will fire the item out of a tube. Items are either fired out at variable intervals exactly as the ground dispenser is implemented, or they can be manually activated. Items are shot out by directly setting the velocity of the rigidbody that is being fired. Originally, we were applying a force to the rigidbody, but found that items of different weights would fly different distances. Due to the shape of the dispenser, not every item can be dispensed through it, as larger items have troubles with getting stuck inside of it.

This dispenser also has the option to modify the direction that the item being dispensed is shot out at. This is currently only used in the level where objects can be sorted into different buckets. Since the item is shot from the back of the dispenser, it will hit the walls of the dispenser if it is not straight out. This is used to randomize the direction of bouncing bombs that are rapidly released from the dispenser.

Sky Dispensers

Sky Dispensers dispense items from above, spawning them outside the view of the camera. The items will then fall into the level. Items are spawned at variable intervals exactly as the ground and wall dispensers are implemented. Items are spawned within two options for a bounding area, either a square or a circle. For both shapes, the center of the bounding area is defined as the position of the dispenser. The square can be defined by a width and depth. Though it is possible to get a uniform distribution for the square, we chose to take the simple approach and select a random x position then a random z position for the item to

be spawned, both within the defined width and depth. This will tend to spawn more items near the center of the dispenser when using this bounding shape. The circle can be defined with a radius. We pick a random angle in degrees and then pick a random distance up to the radius. Using these, we calculate the position to spawn the item.

Hazards

Lasers

Lasers are a hazard that, upon contact with a character, item, or marked piece of environment, dissolves and destroys the object. The laser works by firing a spherecast from its origin to a set length that is always larger than the stage. We want to gather all objects in the path of the laser before the laser hits a non-dissolvable object, like a wall. All objects hit by the spherecast are compiled into a list, whether or not it is dissolvable. We then run through the entire list, looking at the tag of each object hit. If the tag matches a tag that we've set as being a dissolvable object, then we add it to a new list. Designers can optionally set specific objects to be ignored, even if their tag matches a dissolvable tag. As an example, this is used by the Laser Cannon to ignore the cannon and its holder from being accidently dissolved while firing. While running through the list we are also keeping track of the closest non-dissolvable object's distance. After running through the entire list, we run through our new list of dissolvable objects. For each object in this list, if the distance to it is less than the distance to the closest non-dissolvable object, then we mark the object to be dissolved. We also apply a strong impulse to the object, knocking it backwards to add some power to the impact.

Using the distance to the closest non-dissolvable object, we can display the laser hitting that object. The laser is a simple cylinder with a sphere at the end, both with bloom applied. The length of the cylinder is adjusted to scale with the distance to the closest non-dissolvable object. We also apply some small variation to the size and color of the laser at a fixed rate to give it an energized look.

The width of the laser can be controlled externally, as well as whether or not the laser will dissolve anything it hits, allowing for custom implementations of the laser. For example, the lasers in the laser dodging level start off by displaying a very thin laser that warns players not to stand in its path. Then after a short time, the laser grows and is set to begin dissolving. After a few seconds, the laser shrinks back down and disappears. The spherecast is designed to be slightly smaller than the visuals of the laser, allowing players a little more room for error when navigating closely around the laser.

The actual dissolving of objects is done through a custom shader. The shader is a simple dissolve shader with a glow applied to the edges of the dissolve. The width and color of the edge can be controlled by the designer. When an object is set to dissolve, a component is added to it that manages the dissolve. The script compiles all of the rigidbodies and renderers on the object. Our materials have the dissolving display logic built into them. The only thing we need to do is change the render queue of the material to the transparent queue and set the color of the dissolve edge. Then, the dissolve amount is slowly changed to entirely dissolve the object over time. The dissolve is done by using a generic perlin noise texture. The position of the fragment in world space is used for the uv when sampling from the texture. If the value of the texture is less than the current threshold, then the color or the fragment is set to the dissolve color. If

the value of the texture is less than the current threshold plus the width of the colored edge, then the fragment is clipped, making that portion of the object transparent.

The rigidbodies are given a large amount of drag and set to ignore gravity, quickly slowing the object down from the large impulse we applied to it and making them appear to float out of existence.

Portals

Portals are a hazard that suck players in, swirl the players in a decaying orbit, and kill them. Portals have been set up in such a way that other aspects of the environment can open and close them with a public function. They use four main components to perform their functionality. The PortalDoor component is in charge of managing the other components and triggering animations and particle effects. When the door is opening, it activates the SuctionZone component, starts the particle effect, tells the PortalKillZone to activate, and plays the opening animation for the door. When the door is closing, it deactivates those same components, ends the particle effect, and plays the door closing animation.The SuctionZone component uses a trigger collider to detect what objects are within its range. For each object within range, it applies a force towards its focal point. The force becomes larger the closer the object is to the focal point. When active the PortalKillZone detects objects and players entering its trigger area, marks any players as dead, and adds a PortalShrink component to objects that do not already have one. The PortalShrink component detects the state of the portal that created it, shrinks the object it is on, animates the motion of the object it is on, and deletes the object it is on after the PortalDoor closes or after a certain amount of time. The shrinking logic is inherited from ScaleWithJoints, which is a utility class that we use to scale an object

down while accounting for all of the rigidbodies and joints that make up the object. The motion is a simple swirl effect that moves the object in a decaying orbit around the center of the portal. The PortalShrink tracks the state of the door by checking if the PortalKillZone that spawned it is active. A simple float timer is used to determine when to destroy the object if the portal door is still open.

Grinders

Grinders are an additional hazard designed to fill the same space as the acid pools. Players can fall into them and die, either by being ground up or by being pulled into another hazard, such as a lava pit. They were never fully implemented and currently just destroy the player on contact. However, our plan was to set up a system where we would attach a joint to any holdable or player that touches the grinder. This joint would have some break force, giving the players the chance to break free from the grinder. If unsuccessful from breaking out, the player would be dragged to their demise. A simpler version of a functioning grinder can be seen on the conveyor belt level, where players will be sucked into the grinder if they make contact with it.

9 Asset Overview

Super Collider is built to be a simple and goofy game, and our game assets were built to convey that feeling. Though we never explicitly tell the players where they are and what is going on in the game, we used the concept of having a futuristic facility testing subjects in a number of different hazardous scenarios, similar to what one would say when describing Aperture Science from the game *Portal*.

All of our art assets are made up of basic shapes and are flatly colored. Very few textures are used within the game. This decision was in part made because of the limited resources we had for creating art assets but proved to work well with our goals for creating the game. We felt that since we wanted our game to appeal to those with limited experience with video games, having very simple objects would make reading what is on-screen much easier. It is very easy to distinguish weapons and hazards from the static environment.

Our weapons all follow the concept of being prototype weapons. They are all kind of out there in terms of what you would expect from a "normal" weapon. We used *Codename: Kids Next Door* (*KND*), an animated television show from the late '90s, to help us brainstorm and generate ideas for weapons. We would ask the question "What would a weapon from *KND* look like if it were further refined from the very basic materials that the kids had to work with from the show?" For example, we had the idea of a gun that would shoot tennis balls that would bounce all around the room. Tennis balls fit the aesthetic of a

weapon from *KND*, but not so much for our game. We refined the idea and converted the tennis balls into erratic balls of energy that would bounce haphazardly around the room.

The complete list of assets may be found in the Asset Appendix.

10 Play Testing and Results

Playtesting was an impactful part of the *Super Collider* development process. Most of our playtests involved simply allowing players to play the game and recording observations and any feedback the players gave. Throughout this process, players gave a lot of positive feedback and the overall direction of the game never seemed to be an issue. However, there were a lot of smaller implementation issues that became obvious through playtesting. This section will cover most of those.

The first punching mechanic used a hold and release method for charging the attack. After one or two playtests, we received a lot of feedback complaining that this system was too slow and felt non-responsive. In response, we sped up the punching and implemented a system where players repeatedly mash a button to punch and the next punch can start even before the last punch ends. This system worked out well, and our next playtest made it obvious that the changes were a significant improvement.

Some issues were observed with the rocket hammer during playtests. Players that had not used it before would pick it up and press the “attack” button repeatedly instead of holding it down. The rocket hammer was also powerful in close range and the fuel system did not limit the weapon’s usage enough and was confusing without any visual indication as to when it could or could not be used. Based on these observations, we modified the hammer to be a one-use item that locks the player into using it for a short

duration. After that duration, the rocket hammer dissolves. After another playtest, it became clear that these changes improved the game balance of the item and made it more interesting to use.

The portal level is another example of a level that changed a lot based on data from playtests. The portal level features three large portals that can suck players in, killing them. The airlock level was generally considered a strongly designed level in playtests, but we needed to update it to fit in with our new level system and our updated graphics. While making this transition, we removed a few of the minor hazards from around the stage and made the stage slightly bigger. These changes did not improve the experience of the level and that became immediately apparent in playtests, as matches would last for an extended period of time. In response, we shrunk the level down, added in more hazards. Some hazards were the same as their airlock level counterparts, while others were new to the stage. The next playtest showed that these changes resolved the problem and made the level more fun and hectic.

11 Post Mortem

During the course of developing *Super Collider,* many things went well, but we also faced some challenges. The main things that went well were our response to feedback, our communication, our use of clearly defined roles, our culture of experimentation, and our defined set of processes. We made a strong effort to take feedback from playtests and advisors heavily into consideration, and pivoted on many decisions and implementations based on that feedback. This led to a stronger game that more people could enjoy. Our communication with external team members such as composers and artists was strong and there were very few misunderstandings around what needed to be created for a given asset. Within the core team, everyone settled into clear roles with ownership over certain parts of the project. Whenever someone had a question about a given feature, they knew who they could ask. This made it easy to gather information and appropriately assign tasks. We also had a strong culture of trying new things and experimenting. Some experimentation was fruitful and led to positive implementations in the game, while other experimentation gave better insight into the nature of the experience that we were crafting. Both the positive and negative results were used to influence our design decisions. Once we settled on our set of production processes, we achieved a good flow of productivity. We decided on a set of policies early and iterated on them seeing what worked and what did not work for our team. Some of the key policies we implemented were code reviews, in person work days, and our level design pipeline. In person work days in particular proved to have a significant impact on our ability to consistently produce results. The level design pipeline was something we established primarily to add clarity to the current status of any in development level. Being able to say that a level is in a specific, defined stage of development made it a

lot clearer what work still needed to be done on that level. Overall, we produced a good project considering the timeframe and budget.

There were also a few setbacks that we encountered during the project. In particular, current events played a major role in stagnating our productivity by preventing in person meetings and disrupting everyone from focusing on the project. Our main milestone in the middle of the second semester was to finish certain features before the Game Developer Conference (GDC), but when the conference was cancelled, this milestone was mostly disregarded. Early on, we also neglected to consider all of the steps that we would need to take in order to publish the game. As a result, the project is further from being published than we would like. Additionally, while we used a strong set of processes, it took us a significant amount of time to settle on those processes. It was not until the beginning of the second semester that we found the set of processes that worked best for our team.

The next steps for this project are all in service of our continued goal of publishing the game and include the creation of art assets, the creation of additional weapon content, and the implementation of an additional mechanic that would occasionally cause random modifiers to be applied when a round begins. A large number of art, music and sound assets still need to be produced, and we simply cannot publish the game without replacing some of our current test models. We would also like to include a larger variety of weapons to increase the replayability factor of the game. Random stage modifiers were a long term stretch goal that we originally expected to complete during this semester. We still see the value in this mechanic and would like to implement it before we publish the game.

12 Bibliography

Boneloaf "Gang Beasts Debug Gameplay" Youtube, May 8 2015

https://www.youtube.com/watch?v=PI5WCNxghWc

Borderlands, (2009; Frisco, Texas, USA: Gearbox Software, 2K Games), Digital Game.

Burgun, Keith. "Randomness and Game Design." KeithBurgun.net, November 5, 2014.

http://keithburgun.net/randomness-and-game-design/.

Dark Souls, Hidetaka Miyazaki (2011; Tokyo, Japan: FromSoftware, Namco Bandai Games), Digital Game.

Fate/Grand Order, Yosuke Shiokawa, Yoshiki Kanou (2015; Tokyo, Japan: Delightworks, Aniplex), Digital Game.

Ford, Sam "Boneloaf talk Gang Beasts and Unity 5 2015" Youtube, January 17 2015

https://www.youtube.com/watch?v=OzheyWHLWtk

Gang Beasts, (2017; Sheffield, UK: Boneloaf, Double Fine Presents), Digital Game.

Garfield, Richard. "Luck versus Skill." July 10, 2012. Magic Cruise. Video, 55:11.

https://www.youtube.com/watch?v=dSg408i-eKw.

GDC "Gang Beasts at 2014's GDC European Innovation Games Showcase." Youtube, May 12 2015

https://www.youtube.com/watch?v=eFeUyGLOlPA

Genital Jousting, (2018; Cape Town, South Africa: Free Lives, Devolver Digital), Digital Game.

Greenwood, Evan. "The source for the project I made for my AMAZE … Talk." Make Games SA, September 15, 2016.

https://makegamessa.com/discussion/4379/the-source-for-the-project-i-made-for-my-amaze-driving-characters-in-physx-talk

Junya, Motomura. "Guilty Gear Xrd's Art Style : The X Factor Between 2D and 3D" Youtube, May 21 2015. https://www.youtube.com/watch?v=yhGjCzxJV3E

MetallCore999. "Active Ragdoll in Unity." Youtube, March 28 2019.

https://www.youtube.com/watch?v=-50tcHNZsdg

Move or Die, (2016; Bucharest, Romania: Those Awesome Guys), Digital Game.

Ny, Nimso. "How to keep characters upright…" Reddit, Aug 16 2018.

https://www.reddit.com/r/Unity3D/comments/97tqc7/how_to_keep_active_ragdoll_upright_as_seen_here/e4cd4zp/

Pokemon Red version, Satoshi Tajiri (1996; Tokyo, Japan: Game Freak, Nintendo), Digital Game.

Powder, Jeff Lait (2003; Canada: Jeff Lait), Digital Game.

Rayman M, Arnaud Carrette, Riccardo Lenzi (2001; Montreuil, France: Ubi Soft), Digital Game.

Slay the Spire, (2019; Seattle, Washington, USA: MegaCrit, Humble Bundle), Digital Game.

Stick Fight The Game, (2017; Stockholm, Sweden: Landfall Games), Digital Game.

Super Smash Bros. Brawl, Masahiro Sakurai (2008; Tokyo, Japan: Game Arts, HAL Laboratory, Sora Ltd., Nintendo), Digital Game.

Super Smash Bros. Ultimate, Masahiro Sakurai (2018; Tokyo, Japan: Bandai Namco Studios, Sora Ltd., Nintendo), Digital Game.

Swordy, (2016; Auckland, New Zealand: Frogshark), Digital Game.

Terraria, Director by Firstname Lastname (2011; Floyds Knobs, Indiana, USA: Re-Logic, 505 Games), Digital Game.

The Legend of Zelda: Four Swords Adventures, Toshiaki Suzuki (2004; Kyoto, Japan: Nintendo), Digital Game.

Untitled Goose Game, (2019; Melbourne, Australia: House House, Panic), Digital Game.

World of Warcraft, (2004; Irvine, California, USA: Blizzard Entertainment), Digital Game.

Yakuza Kiwami 2, Hiroyuki Sakamoto (2017; Tokyo, Japan: Ryu Ga Gotoku Studio, Sega), Digital Game.

13 Asset Appendix

1. Models
    a. Character Model
    b. Drone model
    c. Environment
        i. Button model
        ii. Conveyor belt model
        iii. Portal shell model
        iv. Portal aperture model
    d. Item/Projectile models
        i. Boxing Glove
        ii. Cactus model
        iii. Kaboomerang model
        iv. Laser cannon model
        v. Plunger model
        vi. Rail gun model
        vii. Rail model
        viii. Rocket hammer model
2. Textures
    a. Controls
        i. PS4 Button textures
        ii. Xbox 360 button textures

    b. Environment
        i. Grate texture
        ii. Warning strip texture
        iii. Conveyor belt texture
    c. UI
        i. Drone icon
        ii. Settings icon
        iii. Menu arrows
        iv. Menu checkbox
        v. Menu check
    d. Noise
        i. Pregenerated Perlin noise
        ii. Bubble distortion
3. Animations
    a. Portal aperture animation
4. Audio
    a. Music
        i. Red Doors (https://opengameart.org/content/red-doors-v20)
        ii. Thrust Sequence (https://opengameart.org/content/thrust-sequence)
    b. SFX
        i. Q009's Weapon Sounds (https://opengameart.org/content/q009s-weapon-sounds)
5. Scenes
    a. Utility
        i. Game

ii. Lobby

iii. MainMenu

iv. Results

b. Levels

i. Arena

ii. Conveyor

iii. Grinder

iv. LaserDodging

v. LaunchingPlatforms

vi. Pitfall

vii. PortalLevel

viii. Sorting

14 Feedback Data Appendix

1st Major Playtest:

- Name Tag on players
- Feedback on hitting or being hit
    - Rumble, sound effect, visual
- Keep suction zones out of button zone
- **Make the controls more responsive**
- Don’t make jump X (make it A!)
- **Punching is very slow and delayed, not very exciting**
- The lobby isn’t intuitive
- Switch up the airlock buttons
- Boxes should activate buttons
- Being knocked out and having control taken away from me is frustrating
- Place more physics objects in front of the airlocks so it conveys that the doors suck you out
- Sound effects

- **Hold to windup or tap to punch**

2nd Major Playtest:

- Trampoline
- Controllable drones
- Better than Gang Beasts
- New game name
- Laser Cannon ammo
- Hazard markers near any place you can fall
- Plunger gun level
- Dragon's Lair arcade game
- Shoving?!
- 2v2

Playtest 3/23/2020

Gameplay

- Need to look at pacing. There are some moments where gameplay gets really slow and I'd like to try avoiding that as much as possible. Thinking about making a "Hurry Up" sort of thing where after some time or after someone dies or something (we can discuss when exactly) something will push players towards an end. Bombs falling from the sky are the most obvious.
- MORE sound effects
- The portal level needs some adjustments. Currently too much safe space around the level. Possibly shrink space, make portals bigger, leave portals always on, etc.
- Decrease time between levels

- Lets players wiggle their arms around while they're holding them up in the lobby and results
- Let players pause even if they're dead (at the very least, let player 1 pause so they can exit if something goes terribly wrong)
- Let players return to the main menu from the lobby
- Pitfall level needs faster/more item spawning
- Rocket Hammer needs to be looked at again. It's lost its mobility.

Bugs:

- Players can still fall out of the laser level. We should probably just add a kill zone around the levels to protect us from this as a general safety net (while also fixing holes like this when we find them)

Playtest 3/31/2020:

Weapons:

- Rocket Hammer
    - Easy to use
    - High mobility
- Liked the disintegration effect
    - Specifically noted that it wasn't as good as the disintegration rifle effect from Jedi Academy (https://youtu.be/w0LtGLkGe5o)

Peculiarities:

- Most effective weapon was the rocket hammer
- Kept blowing themselves up with the bomb gun
- Green gun was performing really well compared to prior playtests
- Couldn't figure out how to use the flail so they just avoided it
- One noted that certain weapons like the cactus grenade and bomb gun would likely work better with more players

When asked for final comments:

- "Turning feels kinda weird, but I think that's the point"
- "It's fun I like the stages"
- One also commented that they wanted to see the "hellish conveyor level" that I had shown them footage of a while back.

On the stages with changes (Portal level and pitfall level):

- They didn't have any trouble killing each other and themselves on either stage
- Some matches on the pitfall level were over before the wall dispensers dispensed any weapons (As a player would kill the other with a weapon from the ground dispenser very early. Usually a bomb gun)

Bugs:

- Cactus seems to be able to block lasers
- Trouble Picking up flail (I think they weren't trying to grab the chain)

Playtest 4/8:

- Players like the changes the Rocket Hammer. I think I want to make it so it will only be dissolved if it was used for more than a second. It looks a little weird when a player uses it, immediately gets knocked down, and it dissolves.
- Tested out the drones that players can control after death. Players liked it - gave them something to do while dead. All the matches were nice and quick. I'll continue with this and polish it up.
- Should look at nerfing the laser cannon. My thoughts are to shorten the shot to a quick burst and make it single-use.