Date: February 24, 2010

Authors: Sela Davis
Chip Hilseberg
Jonathan Lobaugh
Eric Moreau
Nicholas Wilsey

Abstract

In this document, we will discuss the game *Trigger Happy*. We discuss at a high level what the concept of our game is and how it works. Additionally we discuss the business case for making *Trigger Happy* and analyze games, previously released, which are considered our competition, and games that we drew influence from. The high level discussion of our game is followed by a more detailed game design document, laying out what the player will experience while they are playing the game. This game design document will also describe the mechanics behind *Trigger Happy*. Details that are covered include the game world, what that world will look like and how it feels to the player, and to how and what the player interacts with during gameplay. Following the game design document is the art bible. Inside this portion of the document describes everything a player will see while playing the game. It will also describe how the world will look and lay out in the final product. The technical design portion of the document breaks down the individual systems on a design level as well as development level. This portion describes how the game will all be fit together to function. The game production document sets up standards that the development team will follow throughout the development process. These standards help maintain a sense of order that will help streamline the development process. Inside this document is the timeline for how the development will progress over the course of 11 weeks starting from March 8th until roughly May 18th. Individual research topics are included after the game production document and detail the proposed interests for each team member that will be included in the final product.

Elevator Pitch

Imagine yourself in a world where you see a group of clones that look exactly like you. Also imagine that these clones are fighting tooth and nail with contraptions that harness the power of physics itself. You see some clones that are larger sloth-like versions of the other clones. You also see another group of clones up on the ceiling firing rockets down at these sloths. In the far corner you see what appears to be a series of barriers that appear out of nowhere and another clone with a pair of binoculars that fires some sort of projectile right over your shoulder only to hit another clone square in the forehead. You look around and see that you're in some sort of school environment that is attached to a museum and a warehouse.

Welcome to the world of *Trigger Happy*. In this world, players assume the character of clones being exploited in team based matches for the enjoyment of high paying spectators. The clients ordering these engagements can put the teams into a King of the Hill match where the goal is to capture more areas than the opponents, and Assault where one team is attacking and moving a flag across the map through a series of checkpoints while the other team defends. Players have at their disposal a series of weapons that they can take into battle and a combination of contraptions called Modifiers that bring the unpredictable nature of physics into the match. How will you fair? That depends on how well your survival skills are and ability to adapt to any situation that may present itself on the battlefield.

High Concept Document (One-Sheet)

"Kill him again…but this time with feeling."

Short Description: Fight alongside or against your friends in a multiplayer experience that pits one team against the other in dynamic battles. Use physics and science to your advantage to thwart your enemies and achieve victory.

Tone Words: *strategy, action, traps, comical, teamwork*

Game High Concept

Trigger Happy is a 3D first-person shooter (FPS) game that utilizes modifiers to affect enemies' perception of the world around them. The action will take place in a small series of self-contained areas designed to allow plenty of beneficial locations to lay down modifiers in order to both benefit the player and deter opponents. Trigger Happy is team-based, and players will find themselves combining their different abilities together in order to determine the best team makeup. Team sizes can vary depending on player activity at a given time. In addition, players will have the amount of control necessary to find and define their own play styles and play with the weapons they feel comfortable with.

The player's goal in *Trigger Happy* is to help his team win a multiplayer match. Objectives will change based upon which style of match the player chooses. If the match is a king of the hill style, the team's goal will be to enter a region and maintain it by killing all enemies who enter for a set period. If the match is an assault style, the assaulting team's goal is to bring a flag to a predefined set of checkpoints and the defending team's goal is to prevent them from bringing the flag to the final checkpoint for a certain amount of time.

Development Scope

The development scope for *Trigger Happy* will be fairly detailed. The scope will directly reflect how our milestones will be laid out and it is because of this scope that our development process will be laid out using an iterative approach. With a development team of five persons and about six months of design/development time, the project will focus on the first-person shooter genre as well as the player's use of traps within the game world to alter the enemy's perception of the world around them. These two concepts will need to be combined together fluidly to create a dynamic game experience. The development scope and the timeline of milestones will need to reflect that integration of mechanics. The target platform for development will be Microsoft Windows on a PC rather than a console release.

A three-month design cycle with a three-month development cycle is planned. Engine and core systems will be developed in parallel with the design cycle due to their independence from the game idea.

Business Case

Trigger Happy is targeted toward a casual first person shooter audience, which tends to be made up of males aged 18-34. These players have typically played other first person shooters, and many of them are constantly looking around for a new first person shooter experience. *Trigger Happy* will suit their needs by adding unique modifiers, which are remotely triggered game objects a player can throw. The player is given a base set of different modifiers, which they can combine to create varying combinations that suit different play styles.

As the audience is likely familiar with FPSs, they already know what style of gameplay they are getting themselves into with this genre of game. *Trigger Happy* particularly appeals to players of games like *Counter-Strike,* who are looking for a change of pace, as the teamwork aspect will be similar and comfortable. While we are not looking to match *Counter-Strike*'s sales numbers, it is important to note that the *Counterstrike* franchise has sold 9.2 million copies spread across three titles as of December 2008[1].

Games in this genre tend to perform particularly well. However, numbers are often difficult to find, as players of these games tend to purchase on the PC rather than consoles, which are watched much more closely in sales. Games in the genre tend to perform better if they are part of a franchise or series, but original IP and unique ideas certainly can succeed. After all, *Doom*, a seminal game and the title that launched the series, was an original IP. On the other hand, *Wolfenstein 3D*, *Doom's* predecessor, was a spinoff of another genre but did not perform nearly as well.

Five graduate students in the Rochester Institute of Technology's Game Design & Development program will develop Trigger Happy. Each has his or her own particular specialty, and each of these is uniquely suited toward the game. These specialties include important topics such as engine development, 3D modeling and texturing, audio content development, particle systems, and game world design and development. This will help to team to develop a compelling, rich game that will provide many hours of fun.

References

[1] Earnest Cavalli. "Valve Unveils *Half-Life, Counter-Strike* Sales Figures". Internet http://www.wired.com/gamelife/2008/12/valve-unveils-h/, December 3rd, 2008

Game Treatment

Overview

What makes the Future Proficient Solider Academy such a unique place to learn? Is it our status as the first and only armed combat and entertainment school? Is it our groundbreaking work in the fields of *Spawn Camping* and *Performing with Extreme Blood Loss*? Or is it our close ties to bleeding edge R&D labs and hospitals? Maybe it's our industry-trained faculty with real-world experience in a variety of forms, including real weapon historical reenactment, 1 vs. 1000 blood bowls, and customer-designed combat scenarios. All of these things make FPSA an excellent university; but what really makes it unique is its students and their dedication to each other spawn after spawn. As we say here at the academy, "It takes both 'u' and 'i' to spell 'multi-kill.'"

Spots are filling fast for our third freshman class. Apply now and receive a free health pack (clones may apply).

Tone Words: strategy, action, traps, comical, teamwork

Game Concept

Trigger Happy is a 3D first-person shooter (FPS) game that utilizes modifiers to affect the enemy's perception of the world around them. The action will take place in a small series of self-contained areas designed to allow plenty of beneficial locations to lay down modifiers in order to both benefit the player and deter opponents. Trigger Happy is team-based, and players will find themselves combining their different abilities together in order to determine the best team makeup. Team sizes can vary depending on player activity at a given time. In addition, players will have the amount of control necessary to find and define their own play styles and play with the weapons with which they feel comfortable.

Players will be able to roam around maps that are set before a match has begun. A player will be able to pick a combination of weapons from a larger pool to use during gameplay, and they will use these weapons in order to attempt to defeat their opponents. Players will have a certain amount of health, which will allow them to survive one or more shots from weapons. Each weapon will have its own advantages and disadvantages (rate of fire, clip size, etc.). Upon defeating an opponent, the opponent will spawn again in a new location on the map. Trigger Happy will incorporate two game modes that focus on team play and strategy. These two game modes are King of the Hill and Assault. Both modes have different win conditions assigned to them. Once a team has met the win conditions of the round, players will be given the option to begin a new match.

Modifiers will also play a very important role in the player-on-player conflict. Players will have access to all modifier types right from the get-go which the option of augmenting them at various points before and during a match. When a modifier is activated in a given area, rules for an area within a certain distance change for a given time period. For example, placing an anti-gravity modifier will invert gravity for *all players* that enter its bounds. Modifiers are not meant to punish players that enter them, only to change the nature of fighting in a way that can be exploited by the prepared. An anti-gravity

modifier could function as an escape or a kill preventing distraction depending on how prepared a player is to use it. Modifiers fit into many different categories including, but not limited to, physical (anti-grav, slowing) and perceptual (barrier, decoy). Modifiers can also be a part of the environment as global triggers. These modifiers remain in the level at all times and do not disappear.

Particular locations will be designed into the levels to maximize the potential of certain modifiers, but players will not be limited to setting modifiers in these locations. The uniqueness of *Trigger Happy* gameplay comes from the fact that players can change the rules on the fly in local and strategic ways. While a typical FPS only offers one rule set and map over the duration of a fight, our gameplay supports dynamic shifts which means that gameplay is more frantic and player moldable producing a wider range of unique battle experiences.

Critical Path

The flow of the game mainly takes place in a multiplayer arena. Once players boot up the game, they are immediately given the option to go to a game lobby. In the game lobby, a player can either create a new game or join a game in progress. Once in the game, a player is immediately asked to choose a set of weapons and modifiers and is then put on a team and spawns. At this point, the player's goal depends on the game type: either the player must attempt to help his team keep control over a set of points for a set period or the player must attack/defend a number of significant points in the map for a specified amount of time. If the player's health reaches zero, he will respawn again shortly. The player will have the option during this time to reselect weapons and modifiers, but is not required to.

There are two end states to the game for a player: either one team must meet a victory condition, which will end the game, or the player must disconnect from the game. This disconnect, of course, may be player-chosen or may be outside his realm of control. Either situation will bring the player back to the lobby, where he can create a new game, join another game in progress, or exit to the main menu.

Story/Background

Trigger Happy takes place in a future based upon our world. In this future, reenactments (or "re-nacts") and personalized battles have become a popular form of entertainment as well as a lofty career goal. This has all become possible through the advent and spread of cloning, allowing people to fight, die, and live to fight another day. Leagues have begun to spring up to support this sport, and investors and sponsors constantly pay for biggest consumable required: clones. In response, many organizations have been founded. Some support these battles, while others fight to reduce clone waste. One of the most significant of these is the Future Proficient Soldier's Academy, where students learn about combat and weaponry. These students hope this educate will help them get their "big break".

In Trigger Happy, players take control of one of the students of the Future Proficient Soldier's Academy. These students are then dropped into battle with one another and forced to fight in teams for the entertainment of others.

Features and Controls

The unique selling points of *Trigger Happy* lie within the dual combination of allowing the player to choose their weapons to suit their personal play style and using modifiers that can be placed on the

ground or fired at another player. Players will have the ability to spend a given amount of "points" on weapons and a separate amount of "points" on modifiers. Each weapon and modifier will have its own cost based upon its abilities and qualities. This will allow players to strategize effectively as teams while allowing players to enjoy the game in a play style they are comfortable with and adept at.

Development Scope

Since all of us have had experience in development, the number of programmers seem inclusive to all members. However, each member has differing degrees of expertise in programming and the C++ language constructs. With the number of people, the ability for those with development deficiencies can be easily filled in with other members with higher abilities.

Design is handled by a talented designer: Chip is known for his thorough implementations of stories and game worlds. In addition, Nick will be another primary contact point regarding the design. By having two members, we create a game with better group unity without having the design being run by a single member. In addition, the inclusion of two members allows designers to be available for questions at most times when members are working in the lab. Other constructs (email, wiki, etc.) will be managed to help facilitate these collaborative behaviors as well.

Because an original engine will be developed for this game, as with most/all games, the introduction of two members to manage and adjust the engine is quite valuable. Again, much like the designers, having two engine designers' means that work can be divided and the loss of one member does not put the group into instantaneous death spirals. The job of these developers is to build and maintain an engine to fit the needs of the game play programmers. Both developers have a single existing engine that will be utilized in the development of the game.

In the area of game assets, we have three members with skills in the artistic avenue. Nick, Sela, and Eric have media backgrounds and are valuable in the development of game models, sound assets, and art assets. Without such individuals, the game would have very little visual or audio presence. The inclusion of these team members allows us to limit the need for outside sources that might become unreliable or costly.

Finally, since we have worked on projects with DirectX 10 and the use of such a core will be utilized in this project, the introduction of shader programmers helps to stream line the display process and allow basic drawing effects as well as complicated object and post-processing effects to be developed.

Technology Features

There are a number of interesting technologies at play in the background of Trigger Happy. The core of the game utilizes a custom multithreaded game engine, giving the underlying architecture the flexibility it needs to run quickly and effectively. The background music of the game will be generated on the fly in order to help nudge the player's emotion and give them information about what is going on in the battlefield. Models will be brought into the game engine with metadata already attached, simplifying the content pipeline and keeping data with its model. Finally, a unique achievement system will be put in place to keep players engaged and give them goals that persist across play sessions.

Potential Technology Roadblocks

These neat technology features all have a show-stopping potential. As each is pushing the bounds of current research and technology, there is a fallback in place in case a system cannot be successfully implemented -- with the exception of the multithreaded game engine. This engine is a significant risk, but provides massive potential benefit to the users of the software.

Platform

Trigger Happy is targeted at PCs running Microsoft Windows Vista or Microsoft Windows 7. A significant amount of processing power and a hefty graphics card will be required for the game, but the engine will do its best to run smoothly even on machine a couple weeks old. The game will be controlled with a keyboard and mouse, and will not require the number pad on the right side of the keyboard in order to support laptop and small-form keyboards. This release will be targeted at an English-speaking American audience.

Business Case

Trigger Happy is targeted toward a casual first person shooter audience, which tends to be made up of males, aged 18-34. These players have typically played other first person shooters, and many of them are constantly looking around for a new first person shooter experience. *Trigger Happy* will suit their needs by adding unique modifiers, which are remotely triggered game objects a player can throw. The player is given a base set of different modifiers, which they can upgrade to create varying combinations that suit different play styles.

As the audience is likely familiar with FPSs, they already know what style of gameplay they are getting themselves into with this genre of game. *Trigger Happy* particularly appeals to players of games like *Counter-Strike,* who are looking for a change of pace, as the teamwork aspect will be similar and comfortable. While we are not looking to match *Counter-Strike*'s sales numbers, it is important to note that the *Counterstrike* franchise has sold 9.2 million copies spread across three titles as of December 2008[1].

Games in this genre tend to perform particularly well despite the lack of public sales figures. Titles such as *Call of Duty: Modern Warfare* has sold millions. The latest installment, *Modern Warfare* 2 sold roughly 4.7 million in its first 24 hours in the United States and United Kingdom alone[2]. The *Call of Duty* franchise boasts a dedicated fan base on the PC platform that tends to buy most multiplayer FPSs upon release. While *Trigger Happy* will not be as highly publicized as *Battlefield: Bad Company 2*, it will still have the potential to become a sleeper hit. New IP can perform well in this genre, though the biggest hits are nearly always franchise titles. Many FPSs are released every year, but most of them are designed for a single-player experience rather than a multiplayer experience, which is what *Trigger Happy* will provide.

Trigger Happy will be developed by five graduate students in the Rochester Institute of Technology's Game Design & Development program. Each has his or her own particular specialty, and each of these is uniquely suited toward the game. These specialties include important topics such as engine development, 3D modeling and texturing, audio content development, particle systems, and game world design and development. This will help to team to develop a compelling, rich game that will provide many hours of fun. In addition, each has a research project that will provide a solution to a significant challenge, improving the *Trigger Happy* experience and making the game more appealing.

References

[1] Earnest Cavalli. "Valve Unveils *Half-Life, Counter-Strike* Sales Figures". Internet http://www.wired.com/gamelife/2008/12/valve-unveils-h/, December 3rd, 2008
[2] Ben Silverman. "'Modern Warfare 2' breaks day-one entertainment sales record". Internet http://videogames.yahoo.com/events/plugged-in/-modern-warfare-2-breaks-day-one-entertainment-sales-record/1372471

Competitive Analysis

Doom

Doom is considered the seminal FPS, and launched the entire genre into the spotlight. While many of its mechanics feel clunky now, it was a marvel of modern technology back in 1993. It is suspected that over 10 million shareware copies of Doom were installed on computers at the height of its popularity, and it proved to be possibly *the* most influential game to many current developers. Its most significant advantage was the fact that it featured a 3D graphics engine and looked much more visually appealing than everything else. On the other hand, it did not feature such modern improvements as a physics engine, jumping, looking around, etc.

Doom, much like its predecessors, featured the basic FPS mechanics found in *Trigger Happy*: running and shooting. However, *Doom* polished them and packaged them into a gorgeous landscape that appealed to many people. This quickly brought the concept of FPSs to homes and workplaces everywhere, and spawned the creation of many similar games. It is impossible to build an FPS without looking back to *Doom* as a resource and guide, even though most of the mechanics and gameplay features have been significantly improved.

Quake

Quake was the first major FPS playable over the internet (as opposed to over a local area network). *Quake* sales on the PC are unavailable, but the game spawned three sequels in the main franchise and a spinoff titled *Enemy Territory: Quake Wars*. It helped to popularize online play, a significant feature of *Trigger Happy* and most modern FPSs, and was one of the first games to have a series of officially sanctioned tournaments. The *Quake* series is still celebrated today at QuakeCon, which was originally created to bring pros together for these tournaments.

While FPSs have evolved significantly in the past fourteen years, *Quake* is still a seminal game when it comes to multiplayer in the genre. It featured now-standard features such as looking around with the mouse (though it was not standard), jumping, and a client-server model for multiplayer. These were all significant improvements over *Doom*, which used a prior version of the engine and was developed by the same company. *Trigger Happy* uses *Quake* as a respectable source for how to do multiplayer well, but also contains many of these now-standard FPS features as well as a unique "modifier" mechanic.

Unreal Tournament 2004 (UT2K4)

Unreal Tournament is an excellent first example to look at for its FPS elements. While sales data are unavailable online, *UT2K4* is considered one of the best in the Unreal Tournament franchise and performed remarkably in most reviews, giving it an average of 93 on Metacritic. (http://www.metacritic.com/games/platforms/pc/unrealtournament2004) In addition, it was awarded titles such as "Best Multiplayer Game of the Year" in multiple print magazines.

UT2K4 features a fast-paced, frenetic multiplayer death match system that is practically unrivalled (depending on the person you talk to) among PC FPS games. It will act as excellent inspiration and ideally help to guide the design phase. It is considered one of Epic's crowning achievements, and there is little to research in terms of "what went wrong" and more to study regarding "*why* things went well". Even though it is considered one of the best FPS games out there, there is still room in gamers' hearts

for *Trigger Happy* -- FPS players are historically interested in exploring a number of FPS games with varying mechanics. *Trigger Happy*'s modifier mechanics should very easily help to differentiate it from the crowd, and will likely draw in players of *UT2K4* and other games who are looking to experience something different. By appealing to them quickly and showing the strengths of the product, they are more likely to return to play often.

Team Fortress 2 (TF2)

Team Fortress 2 is another well-known FPS released by Valve. While sales data are unavailable, *TF2* is one of the most popular games in the genre now -- and for good reason. It too has won awards from sites such as IGN and 1UP, all of which cite its mechanics, visual style, and comedic elements. *TF2* is proof that the typically serious FPS crowd does indeed have an appetite for comedy, which can easily become a drawing point for *Trigger Happy* as well.

While its predecessors and the name of its developer helped get it out to the public in the first place, it truly shines in the mechanics. *TF2* features two teams that compete across a variety of different maps for goals such as pushing a cart from one end of the map to another, capturing the opponent's flag, or maintaining control over certain points on the map. Each player has the option to choose from one of nine classes, allowing players to experience a variety of different options and adapt themselves based off the team's needs -- something that *Trigger Happy* allows for with its modifiers as well.

There are two specific classes that are applicable in this instance: the Engineer and the Demoman. Each provides the player with the ability to lay something unexpected in the terrain of the world, much like the modifiers in *Trigger Happy*. The Engineer has the ability to place down four different types of machines in the world: a Sentry Gun, which automatically fires at any opposing players in range; a Dispenser, which heals and refills the ammunition of any friendly player that comes near it; and a Teleporter Entrance and Exit, which allows players to unexpectedly (to the other team, at least!) move from one point to another. Each Engineer may only place one of each at a time. The Demoman, on the other hand, has the ability to lay down sticky bombs on the ground. They may then detonate them at any given point in time with a single click of a button, surprising players who do not see them or pay attention to their surroundings. These two classes make playing various maps in Team Fortress interesting, as players must always pay attention to the environment around them. Valve takes this one step further by creating maps with areas that would benefit from a sentry gun or sticky bombs. Player behavior has emerged from this, and players tend to specifically scout out certain areas of the level to check for these unexpected elements or attempt to take different paths all together. Valve has shown that specific level design can make this a positive effect rather than a negative effect.

Shadowrun

Shadowrun is another FPS based off a pen-and-paper RPG intellectual property. It was released for both Windows and the Xbox 360 and includes some interesting mechanics that are worth looking at. Reviews for the game were mixed, and the game received a 66 (Xbox 360)/67 (Windows) on Metacritic. Major criticisms of the title included its high price ($60 on release for the Xbox 360) and its limited number of maps and game modes. In addition, many people were disappointed with the direction the IP was taken in; many players were disappointed to find a FPS rather than an RPG. Despite this, *Shadowrun* sold 400,000 copies on the Xbox 360 and an unreleased number of copies on Windows.

Shadowrun allowed players to purchase abilities at the beginning of a round with money earned through teamwork and kills. These abilities were broken up into two categories: magic and technology. These could each be used by spending a certain amount of "essence", which varied per technique. Players had a maximum amount of essence determined by the race chosen before the match started. This added a unique strategic element to the game and was hailed as one of the most innovative points of *Shadowrun*. Moreover, players had a tendency to strategize ahead of time in order to create a team with a variety of abilities. This is something that *Trigger Happy* should succeed at as well, and this system will be very important for further analysis during the design phase.

Halo 3

Halo: Combat Evolved was the Next Big Thing on the original Xbox, and *Halo3* was the first game in the franchise to hit the Xbox 360. It has sold over 10,000,000 copies during its lifetime, nearly 3,000,000 of which were first week sales in the Americas. It falls into the same genre of FPS with *Trigger Happy*, and focuses on providing a strong, cohesive experience in a unique world. Most importantly, *Halo 3* focuses on a polished multiplayer experience, and there is a significant amount that can be studied in the game. Level design is one of the most important things that Bungie and Microsoft focused on during the development of *Halo 3*'s multiplayer, and it shows. Moreover, they also spent a significant amount of time balancing and tweaking their melee, guns, and grenades. Similarly, *Trigger Happy* will require a lot of time in balancing and tweaking.

Unfortunately, there is very little mechanically that makes *Halo* 3 stand out. Instead, the game has become known for its polish and for being one of the first franchises after *Goldeneye* 007 and *Perfect Dark* to be successful in a console. The modifiers in *Trigger Happy* will help the game stand out from this blockbuster megahit, and will provide it its own place in the world of FPSs.

Call of Duty: Modern Warfare 2

Call of Duty: Modern Warfare 2 is the newest game in the *Call of Duty* franchise and the sixth in the series. It managed to sell 6.97 million copies on the PS3 and 9.71 million copies on the 360. Sales figures are unavailable on the PC, but the market for the game is still there and there is a dedicated community on the platform. *Modern Warfare 2* is a much more realistic game than *Trigger Happy*, and it shows in all of its elements, including its visual style.

More importantly, it focuses on realism in its gameplay and focuses toward a very hardcore player. Moreover, it features a leveling system that mostly appeals to players who want to put in many hours a day rather than an hour every once in a while. *Trigger Happy*, on the other hand, focuses on an audience who has less desire to spend their entire life in the game. Ultimately, *Trigger Happy* only competes with *Modern Warfare 2* due to the genre rather than the target market. The FPS market is large enough that it has split, and *Trigger Happy* and *Modern Warfare 2* are on different sides of the chasm.

Game Design Document

Executive Summary

This document (*Trigger Happy* Game Design Document) describes the gameplay elements of *Trigger Happy*, including all specifications, details, and story elements. Other documents for *Trigger Happy* include the Technical Design Document, Art Bible, and Audio Design Document. This executive summary focuses on the gameplay and story, but touches upon other elements as well.

Trigger Happy is a 3D first-person shooter (FPS) game that utilizes modifiers to affect the enemy's perception of the world around them. The action will take place in a small series of self-contained areas designed to allow plenty of beneficial locations to lay down modifiers in order to both benefit the player and deter opponents. *Trigger Happy* is team-based, and players will find themselves combining their different abilities together in order to determine the best team makeup. Team sizes can vary depending on player activity at a given time. In addition, players will have the amount of control necessary to find and define their own play styles and play with the weapons and techniques they feel comfortable with.

Players will be able to roam around maps that are set before a match has begun. A player will be able to pick a combination of weapons from a larger pool to use during gameplay, and they will use these weapons in order to attempt to defeat their opponents. Players will have a certain amount of health, which will allow them to survive one or more shots from weapons. Each weapon will have its own advantages and disadvantages (rate of fire, clip size, etc.). Upon defeating an opponent, the opponent will spawn again in a new location on the map. Trigger Happy will incorporate two game modes that focus on team play and strategy. These two game modes are King of the Hill and Assault. Both modes have different win conditions assigned to them. Once a team has met the win conditions of the round, players will be given the option to begin a new match.

Modifiers will also play a very important role in the player-on-player conflict. Players will have access to all modifier types right from the get-go which the option of augmenting them at various points before and during a match. When a modifier is activated in a given area, rules for an area within a certain distance change for a given time period. For example, placing an anti-gravity modifier will invert gravity for *all players* that enter its bounds. Modifiers are not meant to punish players that enter them, only to change the nature of fighting in a way that can be exploited by the prepared. An anti-gravity modifier could function as an escape or a kill preventing distraction depending on how prepared a player is to use it. Modifiers fit into many different categories including, but not limited to, physical (anti-grav, slowing) and perceptual (barrier, decoy). Modifiers can also be a part of the environment as global triggers. These modifiers remain in the level at all times and do not disappear.

Particular locations will be designed into the levels to maximize the potential of certain modifiers, but players will not be limited to setting modifiers in these locations. The uniqueness of *Trigger Happy* gameplay comes from the fact that players can change the rules on the fly in local and strategic ways. While a typical FPS only offers one rule set and map over the duration of a fight, our gameplay supports dynamic shifts which means that gameplay is more frantic and player moldable producing a wider range of unique battle experiences.

Setting the Environment

Trigger Happy takes place in a future where war has become obsolete due to vast military expenditures on cloning. However, the resulting improvement in clone production has also given birth to a new form of entertainment based on reckless behavior. This branch of the future TV industry, called Clone Programs, features cloned actors starring in dangerous analogs of everything from game shows, to dramas, to horribly mislead historical re-enactments. The actors in these productions are "soldiers" that are trained both the performing arts and savage bloodshed. In each filming, they play out predefined roles using the weapons and world modifying technology provided by the sponsor of the program. However, when it is all about ratings, the brightest stars bring a style of their own.

Unfortunately, breaking in to the industry is not the easiest thing in the world. Between the expensive clones, the limited opportunities, and the fierce competition, one can quickly find themselves in the unregulated Last-Chance Leagues. Luckily, in recent years, a number of ex-industry professionals and academics founded the Future Proficient Soldier Academy to make the journey a bit easier. At their under-funded relic of an institution (housed in a condemned old fort in the middle of a bustling future city), professors teach all of the Clone Program arts from combat, to tech crafting, to camera work. Their instruction is supported by the healing arts of the militant, environmentalist-lead Appendage Conservation Front and the threadbare funding of the Impossible Possibilities research firm. But with its relentless grit and reckless disregard for safety, the Future Proficient Soldier Academy may one day be the best chance for industry hopefuls across the galaxy.

Game Mechanics

Weapons

All weapons have a limited number of bullets associated with them. Along with max ammunition, each weapon has a limited number of rounds included in their clip. When a weapon's clip runs out, the weapon is required to reload. Ammunition is generalized across all weapons so it limits the need to find ammo for a specific weapon. Players pick up ammo boxes that are located through the level to replenish ammo for all their weapons.

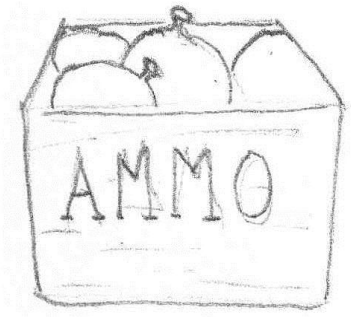


Figure 1: Ammo Box

Players will be allowed to choose 2-4 weapons from a pool of eight to take onto the battlefield. When choosing weapons, players will be given five weapon points to use to select a new combination. These weapons can be chosen at the beginning of the game or when they are ready to respawn onto the battlefield. Weapons have been tiered based on damage and other attributes to ensure that there is no completely overbearing combination that players are using during the game. The three tiers are worth 1, 2, and 3 points in the order of lowest to highest. The set of weapons a player takes onto the battlefield can change but will only go into effect when the player respawns again.

Weapons Types

Nullifier Gun

The Nullifier Gun is a secondary weapon that allows players to render Modifier effects null and void for a limited amount of time. The gun itself is a single-shot weapon that fires a high-heat nullifier effect

burst. This burst can also remove any effects that affected players have already received from Modifiers. When firing the gun, players can hold the firing command to fire a longer distance shot. When the shot is fired, regardless of distance, the round will explode and an energy burst releasing the nullifier effect will appear on the screen very briefly. Any player that encounters this energy burst will have the nullifier effect applied to them.

Attributes

Maximum Ammunition	15
Rounds per Clip	3
Fire delay between shots	0.8 seconds
Damage per round	N / A
Range	1-10 meters
Resource Point Allocation	1
Effect Duration	5 seconds

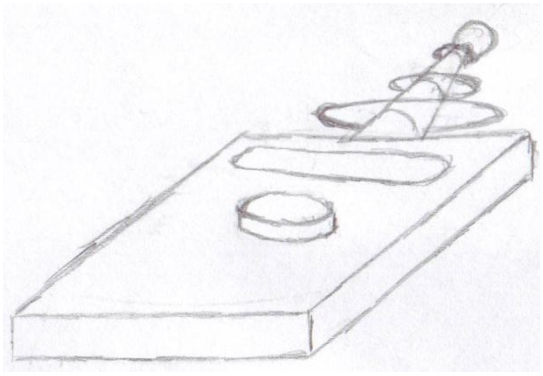


Figure 2: Nullifier Gun Concept

Grenades

When players are stuck in the trenches and need a way out, grenades are the way to go. Grenades are little exploding balls of death that players throw towards enemies to inflict damage on them. The biggest distinction between grenades and other guns is that they are thrown, not shot. These grenades can be banked off walls and thrown into areas that normal bullets cannot reach via direct line of sight. Once a grenade is thrown, it has an internal timer for when it goes off. Once the timer reaches zero, the grenade explodes. Like rockets, the grenade has an explosion radius, which inflicts additional damage to close-by targets. The power of the throw by the player is dependent on the amount of time the fire command is held.

Attributes

Maximum Ammunition	8
Fire delay between throws	3 seconds
Damage per Round	20
Range	30 (max, power of throw dependent)
Resource Point Allocation	1

Figure 3: Pop Can Grenade Concept

Pistol

The pistol is the reliable fallback weapon for most players. It is a compact and accurate single-shot firing weapon with two functions. It can be a single fire weapon or the player can hold down the fire command to charge the shot. While charging the weapon causes the next shot to deal more damage to an enemy, it causes the weapon to overheat faster.

Attributes

Maximum Ammunition	48
Rounds per Clip	6
Fire delay between shots	0.3 seconds
Damage per Round	5 (single shot), 40 (Maximum Charge)
Charge Time	2.5 seconds
Reload Time	1
Range	20 meters
Resource Point Allocation	1

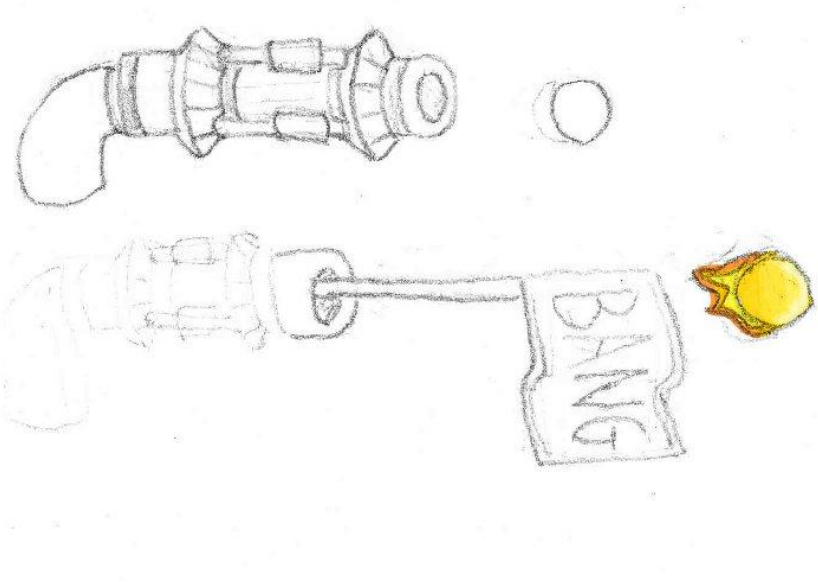


Figure 4: Pistol Concept

Flamethrower

The Flamethrower is the ultimate "close encounter with a bad time" weapon. The flamethrower is a short-range spray weapon that when an enemy encounters the weapon's projectile, they are inflicted with a burning effect that lasts with the target for a fixed number of time. Players wielding this weapon can continue to fire the weapon at targets to inflict more damage and hopefully end their target's existence. This weapon is most effective when the player is around multiple enemies and is looking for a way to turn the tide of battle.

Attributes

Maximum Ammunition	120
Fuel per Clip	35
Fire delay between shots	0.0 seconds
Damage per round	1
Spray Radius	0.5 meters
Damage over time effect after contact	15 seconds
Reload Time	2 seconds
Range	2 meters
Resource Point Allocation	2

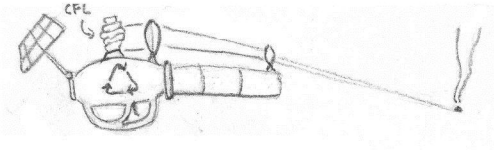


Figure 5: Flamethrower Concept

Shotgun

The shotgun is a weapon that players fire once and spray multiple bullets over a small area at one time. The shotgun is ideal for the close combat fast reaction players who like high damage in small amounts of time. The damage dealt by the shotgun is directly proportional to the number of bullets that encounter the target at a given shot. On a per-fire basis, the shotgun has a higher heat value than that of the assault rifle due to its firing nature. The shotgun also features a slight knockback force with its bullets. Players firing the shotgun can enjoy watching their enemies flung backwards when they are hit with the spray of bullets.

Attributes

Maximum Ammunition	35
Rounds per Clip	8
Fire delay between shots	0.75 seconds
Damage per Round	5
Bullets Fired per Shot	5
Spray Radius	0.5 meters
Reload Time	0.75 seconds
Range	8 meters
Resource Point Allocation	2

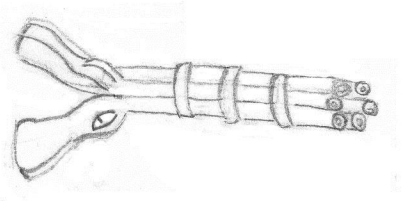


Figure 6: Shotgun Concept

Assault Rifle

The assault rifle is a close encounter with a painful kind. This weapon fires off more rounds before overheating than any other weapon at the player's disposal. This weapon excels at rapid-fire bursts from the player to deal sufficient amounts of damage to their enemies. Each bullet has a low heat value, which allows players to fire an onslaught at their enemies. This weapon is ideal for players who prefer closer encounters with their enemies.

Attributes

Maximum Ammunition	200
Rounds per Clip	30
Fire delay between shots	0.1 seconds
Damage per Round	4
Bullets fired per Shot	1
Reload Time	1 second
Range	15 meters
Resource Point Allocation	2

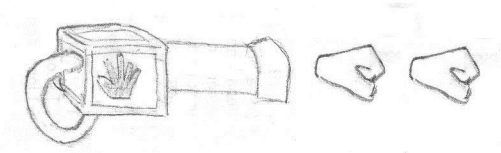


Figure 7: Assault Rifle Concept

Rocket Launcher

The rocket launcher is a cannon with a chip on its shoulder. This weapon is for players that like to see things go boom in a hurry. The rocket launcher fires a rocket in the player's intended direction with fair amount of accuracy. Upon the rocket's contact with a player or surface, it explodes in flaming glory inflicting damage to any enemy within its blast radius. When the rocket explodes, on top of inflicting damage to enemies, the impact of the rocket throws enemies back. The amount of knockback that is incurred from the rocket blast depends on the enemy's proximity to the blast. The rocket launcher is a single shot weapon that has a high resource point allocation so players using this weapon will need to be frugal with how they spend their points to supplement this weapon.

Attributes

Maximum Ammunition	8
Rounds per Clip	1
Fire delay between shots	1.5 seconds
Bullets fired per Shot	1
Blast Radius	0.5 meters
Reload Time	1 second

Range	45 meters
Resource Point Allocation	3

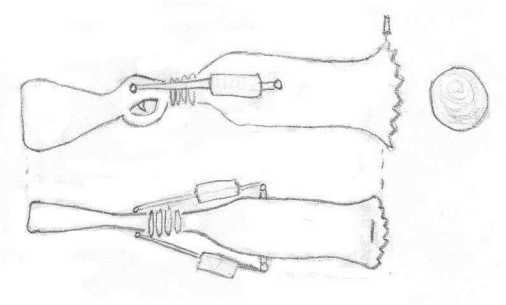


Figure 8: Rocket Launcher Concept

Sniper Rifle

The sniper rifle is meant for long-range single fire precision and pain. Players utilizing this weapon generally will not be in close quarters with their enemy but instead perched far behind the battle picking off their enemies one by one. The sniper rifle is a high impact long recharge weapon that has a zooming scope to allow players better view of their targets from long ranges away. Players using this weapon must be aware of its large heat value that comes with every shot so being efficient with ammunition is key for this weapon.

Attributes

Maximum Ammunition	12
Rounds per Clip	1
Fire delay between shots	1.5 seconds
Damage per Round	70
Bullets fired per shot	1
Reload Time	3 seconds
Range	50 meters
Resource Point Allocation	3

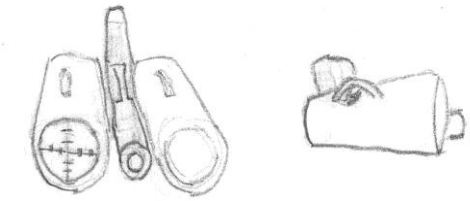


Figure 9: Sniper Rifle Concept

Modifiers

Modifiers are another tool for players to use during gameplay designed to throw off and inflict pain on their enemies and take an active role in shaping the way battles are fought. They are devices that players can place in the environment and trigger them remotely at any point during the match. When a Modifier is triggered, it applies an effect to anyone within its radius. The effect remains attached to players for variable durations depending on what type of modifier was triggered. There are eight

different Modifiers that players have at their disposal to create combinations with. Players are allowed to create combinations of three modifier types.

Each Modifier has an energy value associated with it that is part of a greater energy pool linked to the player. Energy is consumed by a player whenever a Modifier is deployed into the world. The amount of energy consumed by a Modifier is dependent on what type of Modifier the player is deploying as well as the upgrade tier the Modifier is currently set at. Modifiers that are at higher upgrade tiers cost more energy to deploy into the world. This energy is not recoverable until a Modifier is either triggered by the player or is destroyed by an enemy. There is no way for players to regain energy or add to their supply except for the triggering or destruction of their Modifiers. Each Modifier has three tiers of effectiveness that allow for stronger results acted on the players. The higher tiers have increased attributes that will allow for these stronger effects. The other side of the coin is that the higher tiers cost more energy to deploy.

All Modifiers have health attached to them. The health of the Modifier is directly related to the tier that it was deployed as. Higher tiers have higher health values. Players can take out enemy Modifiers by shooting at them. When Modifiers are destroyed, energy is restored to the player that deployed it as if the Modifier was triggered. Effects do not trigger when a Modifier is taken out by players, they just disappear from the environment.

Deploying and Triggering

In order to deploy a Modifier, players will need to equip their Modifier gun and select which type they will wish to deploy. If a player has a weapon equipped at the time, the Modifier gun will automatically come up in place of the weapon with the desired type ready to fire. When firing the Modifier gun, players treat the firing trajectory of the Modifier similar to how grenades would be thrown. When a Modifier is fired, it sticks to the first surface it finds. To deploy higher tier Modifiers, players will hold the fire command down until the desired tier is reached and release the command to deploy it. When holding down the command, a gauge will appear on the screen indicating the tier that the Modifier is being charged to.

When a Modifier is deployed into the world, players will see a target reticle on their screen that is linked to the position of the Modifier. This reticle will have an icon indicating the type of Modifier the player deployed so they have an idea as to what they are triggering. In order to trigger the Modifier, players will need to line up their target reticle with the Modifier's reticle. When both reticles are lined up, players can trigger the Modifier using the right click of their mouse. Players can trigger Modifiers regardless of which weapon they are using. The reticle will have a proximity indicator along with it so players know when another player is within the effect radius of the Modifier should it be triggered. The proximity alarm does not indicate which team the player that is within range is on.

Modifiers as part of the Environment

Special Modifiers are placed inside each level that help add another level of depth to the environment. These Modifiers are special because they are always present in the level and cannot disappear. The effects applied from these Modifiers are permanent until a counterbalance has been met. These Modifiers are meant to open up paths in the environment that the player would normally not consider and offer up new strategic possibilities to teams. The modifiers that are placed in the level are not meant to be beneficial or detrimental towards any team but merely meant to give the team a new outlook on playing the environment.

Modifier Types

Inverse Gravity

The Inverse Gravity modifier is the Modifier of choice for players looking to really throw their enemies for a loop. This Modifier does exactly as the title suggests. When a player triggers this Modifier type, any enemies within its effect radius have their field of gravity instantly reversed. This can be very effective in throwing off team strategies on both offensive and defensive stands since it alters an enemy's plan of attack. When the Inverse Gravity Modifier is triggered, players are thrown upside down and depending on the tier of the Modifier, a force is applied to that person throwing them at the ceiling. Along with inversing the gravity of the player, falling damage is dealt to the affected player. The amount of damage dealt is directly linked to the force applied to the player as well as the height of the fall from where the Modifier was triggered. Falling damage is also dealt when the affect wears off making this Modifier a very painful experience for victims.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	80 points	115 points	140 points
Effect Radius	1.0 meters	1.2 meters	1.5 meters
Effect Duration	13 seconds	15 seconds	18 seconds
Energy Use	22 units	35 units	40 units
Extra Force	10 Newtons	20 Newtons	30 Newtons

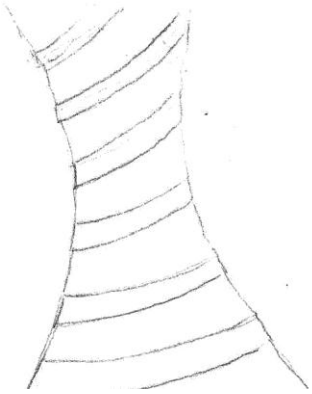


Figure 10: Inverse Gravity Effect Concept

Knockback

The Knockback Modifier is the perfect Modifier for players looking to clear a path or throw their enemies out of the way. The Knockback Modifier when triggered exerts a force on enemies within the effect radius. The force exerted is dependent on the tier the Modifier currently is set at, the higher the tier, the more force exerted. The force pushes players in the opposite direction the player is in relation to the Modifier itself. The impact of the force on players can deal damage depending on how hard and far the force pushes the player back.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	95 points	115 points	135 points
Effect Radius	0.7 meters	1.0 meters	1.3 meters
Effect Duration	Instant	Instant	Instant

Energy Use	19 units	25 units	33 units
Extra Force Applied	10 Newtons	20 Newtons	30 Newtons

Figure 11: Knockback Effect Concept

Increase Gravity

The Increase Gravity Modifier alters an affected player's local gravity. By increasing the gravity, affected players feel as though they are on a different planet. Movements done in normal gravity all of a sudden have a different meaning in a gravity-increased environment. Applying a downward force to players causes them to take damage as they walk and jump.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	75 points	105 points	130 points
Effect Radius	1.2 meters	1.35 meters	1.5 meters
Effect Duration	10 seconds	14 seconds	18 seconds
Energy Use	25 units	33 units	40 units
Downward Force Applied	10 Newtons	22 Newtons	34 Newtons

Figure 12: Increase Gravity Effect Concept

Accelerator

The Accelerator is an invisible ice rink disaster waiting to happen. Players utilizing this Modifier are given the gift of causing their enemies to reduce control of their character for a brief period through the means of high speed and low friction. When a player triggers this Modifier, enemies within its effect radius suddenly experience an increase in speed as well as a lowered friction level. This causes players to lose some control of their character's movements.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	85 points	105 points	120 points
Effect Radius	1.1 meters	1.2 meters	1.5 meters
Effect Duration	10 seconds	14 seconds	17 seconds
Energy Use	19 units	25 units	35 units
Acceleration Amount	3.5x Speed	4.2x Speed	5.0x Speed
Friction Reduction Amount	2.5x Reduction	3.5x Reduction	4.5x Reduction

Figure 13: Accelerator Effect Concept

Barrier

The Barrier Modifier is a multi-purpose Modifier that can be used for both beneficial and detrimental effects. When the Barrier Modifier is triggered, it creates a wall that shoots up perpendicular to the surface that the Modifier is stuck to. Depending on the platform's orientation, players can use the barrier as a platform to jump on to get to higher parts of the level that would not normally be accessible via normal jumping. If timed right, players can use the Barrier Modifier as a catapult to launch players away. The Barrier itself can block bullets and player movement like any normal wall would.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	100 points	135 points	150 points
Effect Radius	N / A	N / A	N / A
Effect Duration	13 seconds	15 seconds	18 seconds
Energy Use	24 units	30 units	38 units
Extra Force	10 Newtons	20 Newtons	30 Newtons

Figure 14: Barrier Concept

Grow

The Grow Modifier is another Modifier that alters player's perception of the world around them. When triggered, the Grow Modifier causes players caught in the effect radius to grow to unnatural sizes. Player's caught with this effect will find themselves having a hard time traveling through the environment since doors and other gaps do not accommodate really large people. There is a benefit to being larger though. When players are larger than normal size, they take less damage. Along with the damage reduction, there is also a slight speed reduction that players incur when they are affected.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	90 points	110 points	130 points
Effect Radius	0.5 meters	0.7 meters	1.0 meters
Effect Duration	5 seconds	7 seconds	9 seconds
Energy Use	15 units	19 units	23 units
Size Multiplier	1.5x	2.0x	2.5x
Damage Reduction	0.8x	0.75x	0.6x
Speed Reduction	0.85x	0.75x	0.65

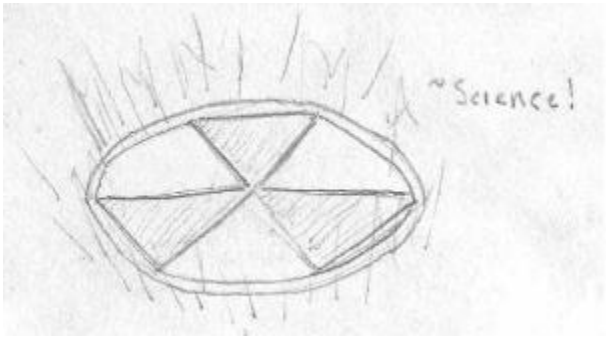


Figure 15: Grow Effect Concept

Shrink

The Shrink Modifier is the exact opposite of the Grow Modifier. This Modifier type causes players to shrink to tiny sizes. This creates an advantage for the deploying player since it forces the affected players to rethink their movement and attacking strategies. Players under this affect take more damage

as they are shot. The upside to being shrunk is that players take on a slight speed increase allowing for faster movement due to their smaller mass.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	90 points	110 points	130 points
Effect Radius	0.5 meters	0.7 meters	1.0 meters
Effect Duration	5 seconds	7 seconds	9 seconds
Energy Use	15 units	19 units	23 units
Size Decrease	0.75x	0.65x	0.5x
Damage Increase	1.15x	1.25x	1.35x
Speed Increase	1.2x	1.35x	1.5x

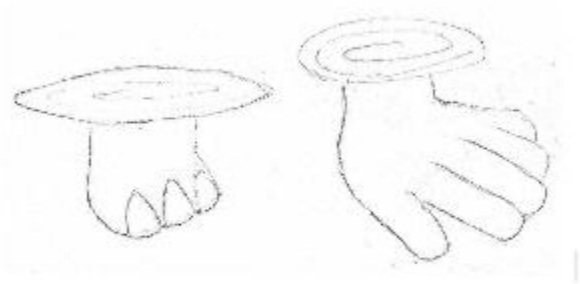


Figure 16: Shrink Effect Concept

Illusion

There are times where players need to create a distraction for the enemies so they can get the jump on them. The Illusion Modifier will do just that. When players trigger the Illusion Modifier, it deploys a series of clones of the player who deployed the Modifier. These clones are meant to confuse the affected players into thinking there are more enemies on the battlefield than there really are. This distraction can give players the opportunity to sneak behind an enemy while they are distracted and finish them off. The decoy itself has health that can be knocked off by gunfire thus neutralizing it permanently.

Attributes

	Tier 1	Tier 2	Tier 3
Modifier Health	100 points	115 points	140 points
Effect Radius	N / A	N / A	N / A
Effect Duration	14 seconds	17 seconds	25 seconds
Energy Use	14 units	18 units	25 units
Illusion Image Number	2 images	4 images	7 images

Selecting Weapons and Modifiers

Before a game starts, players are given a setup period in which to familiarize themselves with a level and plan their weapons and modifiers out. When this setup period occurs, players will be brought to the

weapons and modifiers selection screens. When they have chosen their desired combinations, they can move on. They can change their sets as many times as they choose before the setup timer reaches zero and the game begins. The last set of weapons and modifiers chosen when the period ends is the setup that the player will start battle with. It is to the player's advantage to have a general idea of what combinations they wish to work with from the get-go so they can plan routes and find the best places to lay down their Modifiers.

During a game, players may find that the combination of weapons and / or modifiers they are using is not working against their enemy. This can be due to having a combination that does not counter well against an enemy's combinations. When a player waits to respawn, they are given the opportunity to change their combinations out for a new one. During this wait, they have access to the weapon and modifier selection screens to choose new combinations to take into battle.

Player

Movement and Attacking

The characters players represent in the environment exhibit a bipedal human movement model. This model allows players to move forward, back, and turn side to side. Along with the basic movements that the bipedal model offers players, the characters will be able to jump, crouch, and strafe from side to side.

Shields

The shield is the player's primary life force in the world of Trigger Happy. Players will need to maintain their shield level as they participate in battles. As players take damage, their shield will drop. When the shield runs out, players are completely exposed and the sheer impact of a single bullet, explosives, or other damage dealing objects will kill the player. Damage done to shields can be recovered however. If a player is not engaged in battle for at least 15 seconds after they took damage last, the shield will begin to regenerate at a pace of 3 points per second. If a player starts taking damage again then the regeneration will stop and a new cycle will need to begin.

Death and Respawning

When a player dies, they will be required to wait 10 seconds until they are allowed to respawn again. During this time, players will be given the opportunity to alter their weapon load-out and / or modifier load-out. While waiting for the 10 seconds to count down, players will be brought to the weapon and modifier selection screens. This is where the players can change their load-outs for weapons and modifiers should they choose to. At any point after the 10 seconds, players can proceed to respawn.

Game Modes

All game sessions are played online through team-based matches, in one of two game modes. Players choose the game mode at the beginning of a game session, and cannot change it during a match. During this period, teams have limited access to the game environment. This ensures that teams do not interact with each other before the match begins. Regardless of game mode, the setup period is 1

minute. When the setup period ends, safety gates strategically placed in the level will drop and the game can begin.

Assault

Assault is an attack / defend style game mode, where one team (the attacking team) tries to move its flag into the other's base, while the other (the defending team) tries to stop it from doing so. The game is timed, and if the attacking team successfully moves its flag into the opposing team's base before the time runs out, it wins the game. However, if it fails to do so, the defending team wins. The map is segmented into a series of intermediate checkpoints, however, and before reaching the defending team's base, the attacking team must pass through each one. Successful passage through each adds additional time to the game time. Players who die during Assault are forced to spawn back at their team's base point.

To move the flag through the map, a player from the attacking team must be carrying it. If the player with the flag dies, the flag falls to the ground where that player died. The flag remains in that spot until either a player from the attacking team picks it up, or a player from the defending team interacts with it. If a player from the defending team interacts with it, it is teleported back to the attacking team's last claimed control point (or the attacking team's home base, depending upon the attacking team's progress).

King of the Hill

King of the Hill is a game mode that centers on capturing and defending a region within a level for a fixed amount of time. To capture a region, a team must enter a defined region's boundary and defend it from the other team for a specified amount of time. The first team to have any player in such a region is charged with defending it, and as long as it is only their teammates within the region during the predefined time, they will capture it. Capturing a region adds one point to that team’s total. However, if any member of the other team enters the region before it is captured, the timer pauses, and the defending team must eradicate the intruders before it will resume. If the intruders are able to remove all members of the defending team from the region, their team will then become the defenders. Any time a region has a new defender, the timer resets. The game ends when a team has successfully captured the number of Hills that was set before the match begins. When a player dies during a King of the Hill match, the respawn is a random location away from the battle.

Game States and Flow

The states of the game should be straightforward in their navigation. Players going through each state should be able to clearly understand where they are in the flow of the game. Going from state to state should be quick and clean while reflecting the comedic theme of the game. Moving between the screens should include clean animations and transitions of user interface elements to add to the user's overall experience.

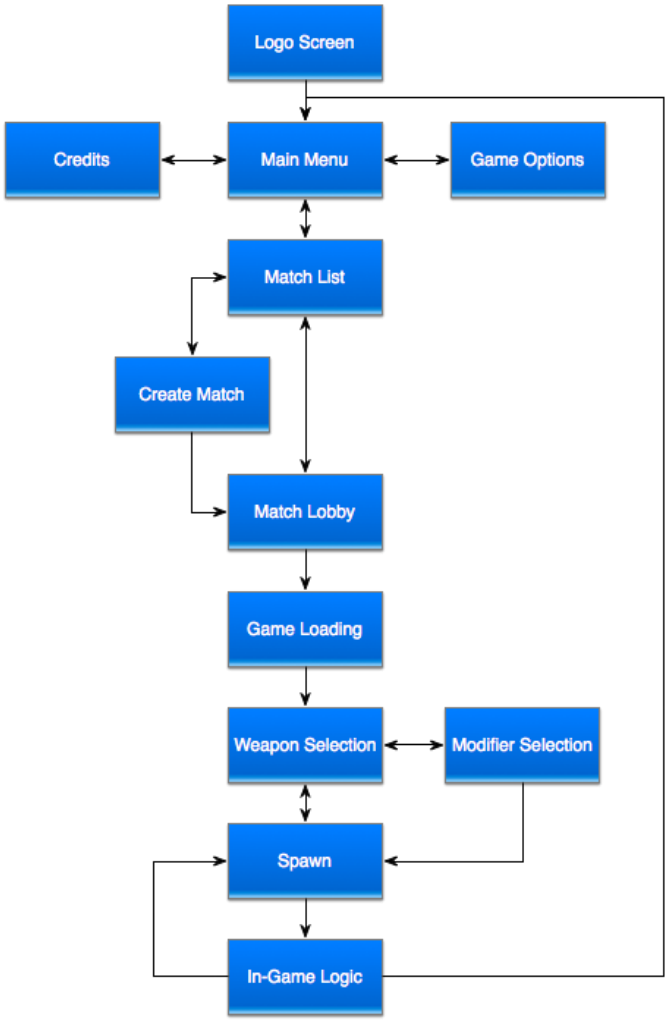


Figure 17: Flow chart of game states

Logo Screen

The logo screen is the first screen players see when they start *Trigger Happy.* The logo screen is a series of fade transitions that show the development team's logo, the Singularity engine's logo, and any copyright details. The logo screen should also contain a brief mention of RIT's GDD graduate program. The purpose of this screen is to give the player eye candy to look at while the background processes of the game load. When *Trigger Happy* has loaded, the logo screen will fade out to the main menu.

Main Menu Screen

The main menu is the first intractable screen the player has access to. Here the player will make their first series of decisions for how they wish to proceed. On this screen we will see the title splash screen along with a menu the players can scroll through. The menu will have all options visible all the time and the option the player is currently selected on will be highlighted by changing the button's background color slightly and having a smooth flash to it.

The options presented to the player will be: go to lobby, access game options, view the game credits, and exit the game.

Figure 18: Main Menu Concept

Go to Match List

Going to the match list brings the player to the game lobby. Here the player will be able to join games already started or create a game of their own if they choose to. The transition from the main menu to the match list consists of the Go to Match List menu item followed by a delayed move of the remaining menu items sliding in the same direction.

Game Options

In this state, players will be able to adjust a number of game options. The transition from the main menu to the game options screen consists of the Game Options menu item sliding to the left first followed by a delayed move of the remaining menu items sliding in the same direction off the screen followed by a fade to black.

Credits

The transition from the main menu to the game credits screen consists of the Credits menu item sliding to the left first followed by a delayed move of the remaining menu items sliding in the same direction off the screen followed by a fade to black.

Exit

When the player goes to the exit state, they are brought to the credits screen. The transition from the main menu to the game credits screen consists of the Exit menu item sliding to the left first followed by a delayed move of the remaining menu items sliding in the same direction off the screen followed by a fade to black.

Game Options

The game options screen allows players to adjust various graphical and performance options pertaining to the game. When leaving the game options screen, the screen fades to black.

Match List

The match list will be the hub in which players will make the choice to either start a game of their own or join an already existing game. This screen will populate with a list of games that are waiting to be filled by other players. Information for these games will include what game mode the creator has chosen, the number of players already in the game, and how many open slots there are left. There will also be an option for players to create their own games. A refresh button will be placed next to the "create game" and join game buttons to refresh the list of matches that are looking for players. Transitions to and from this screen will be simple fade-outs to black.

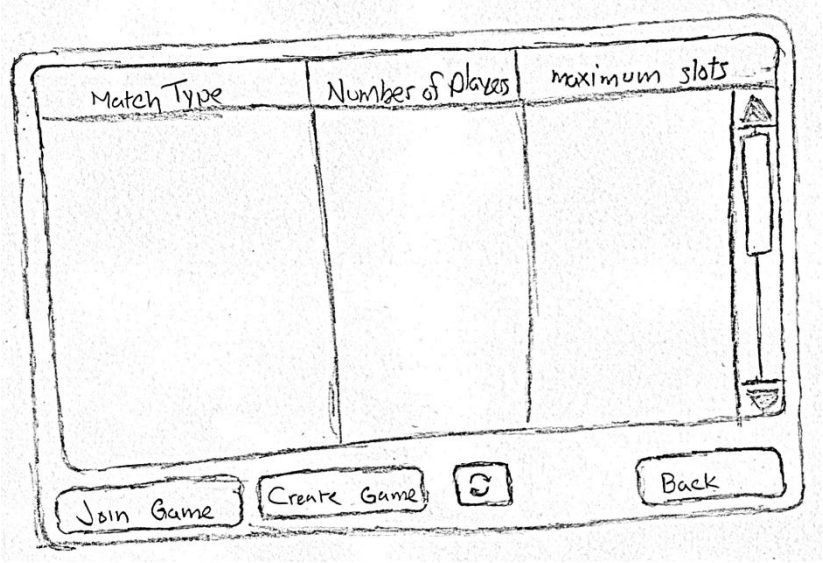


Figure 19: Match List Concept

Creating a Game

Creating a game will bring players to the game creation room. Here they will be able to select the game mode and the max number of players they wish to participate in the game. Secondary options for the potential match include the adjusting the time limit for Assault and the number of captures teams are required to get to win the match. When players are finished setting up the match, they proceed to the match lobby screen which is the screen that players looking to join a game will see.

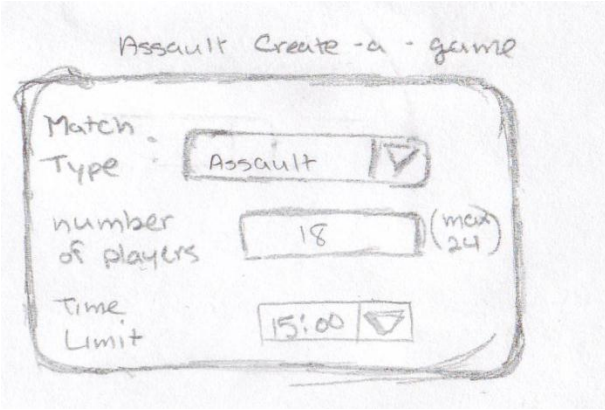


Figure 20: Create Game Concept - Assault

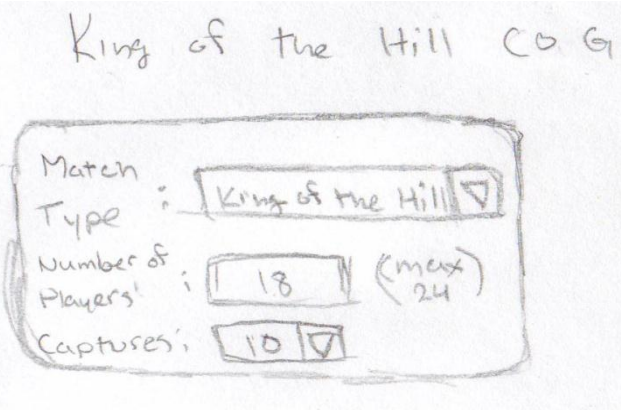


Figure 21: Create Game Concept - King of the Hill

Match Lobby

When players choose a game they wish to join from the game lobby, they are brought to the match lobby screen. Here they will see information about the match that the game creator has set up, panels designated for each team, and any players that have filled these slots. Incoming players will be allowed to choose which team they wish to participate on as long as there are slots for that team available. When players are ready to proceed, they toggle the "ready" button signifying their ready status. When all players have signified that they are ready to play, the game will proceed to the loading screen state.

Game Loading

The game loading screen is a transition screen that goes from match lobbies to the in-game state. On this screen, splash screens offering hints and possible tactics to players. In the background, the game loads its assets and sets up the game session. At the bottom of the screen, there is a loading bar indicating the progress of the loading process so players have an idea as to how long the game has left to load. When the game is finished loading and preparing, the screen fades out and is brought to the Weapon and Modifier selection screens.

Weapon Selection

The weapon selection state happens at two different points during a game. The first situation is when the game initially starts up. When a game is finished loading, players are immediately brought to this screen. The second situation occurs when a player has died and is waiting to respawn. During the respawn period, players have the option to go to the weapon selection screen where they redo their combination of weapons.

On this screen, players are shown all eight weapons represented by icons in a row on the top of the screen. Choosing weapons can be done by both a click of the mouse or by pressing the hotkey assigned to them. Along with the weapon information, there is a row of numbers below the weapon icon that represents the number of that weapon that total the number of teammates using that weapon.

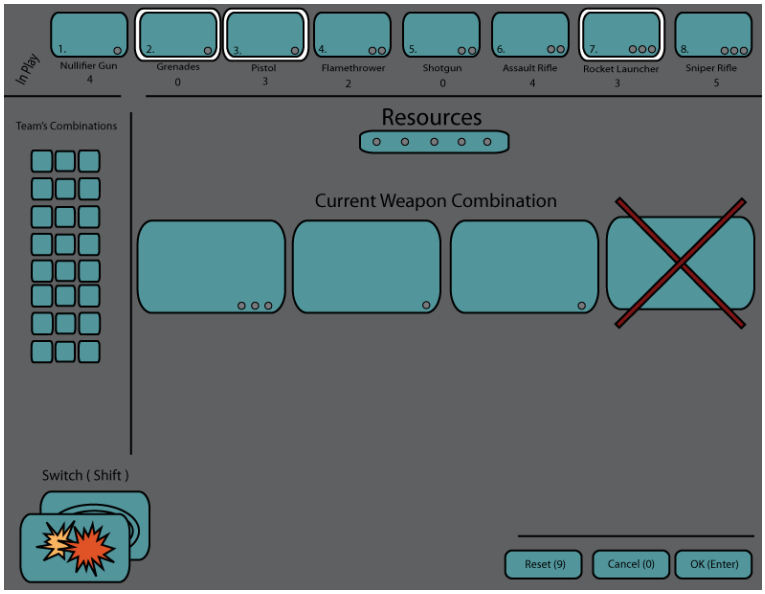


Figure 22: Weapon Selection Screen Mock-up

Other information on this page shows the progression of the player's selections. Weapons that players have selected are highlighted with a white border along with larger icons filling a row of four

slots that players can possibly use. A bar shows the resource points that players have left to use on their weapons. When weapons become unavailable to use based on the resource points they use, a red "X" is placed over the weapon icon in the selection row. A red "X" is also placed over any slots that become unavailable due to combinations that players may come up with. If a player wishes to deselect a weapon, they click or press the hotkey associated to the weapon one more time to deselect it.

Also on the left side of this screen is information about the combination of weapons that players are using. This will help with making informed choices for players as they try to plan a counter strategy. On the bottom right side of the screen are buttons that allow players to reset, cancel, and accept the combination of weapons they wish to use. When a game session is first beginning, players will be forced to choose weapons and modifiers so the "cancel" button will be grayed out. Also, players will not be able to accept their weapon combinations until they have used all of their resource points. Until that criterion has been met, the "ok" button will be grayed out. By clicking the "reset" button, players will reset their weapon selections and all resource points back to the maximum so they can choose new weapons. On the bottom left of the screen is a toggle button that switches between the weapon selections screen and the modifier selection screen.

There will be a cross-fade transition going from the weapon selection screen and the modifier selection screen. Because the layout for both screens is identical, this transition will be easy for players to identify with and allow for speedy selections on either screen.

Modifier Selection

The only way to get to the modifier selection state is toggling from the weapons screen. The modifier selection screen is reached by toggling from the weapon selection screen. On this screen, eight icons represent each modifier with corresponding hotkeys along the number line on standard keyboards. A number is placed under each modifier button that shows how many of that modifier type are being used by teammates. This allows players to make informed choices about what combination they will take into battle.

In the middle of the screen are the three slots that players have to choose modifiers with. Since there is no resource linked to choosing modifiers like there is with choosing weapons, the panel that houses resource points on the weapon screen is not found on the modifier screen. Along the left side of the screen are teammate’s modifier combinations allowing players to make even more informed choices about their selections.

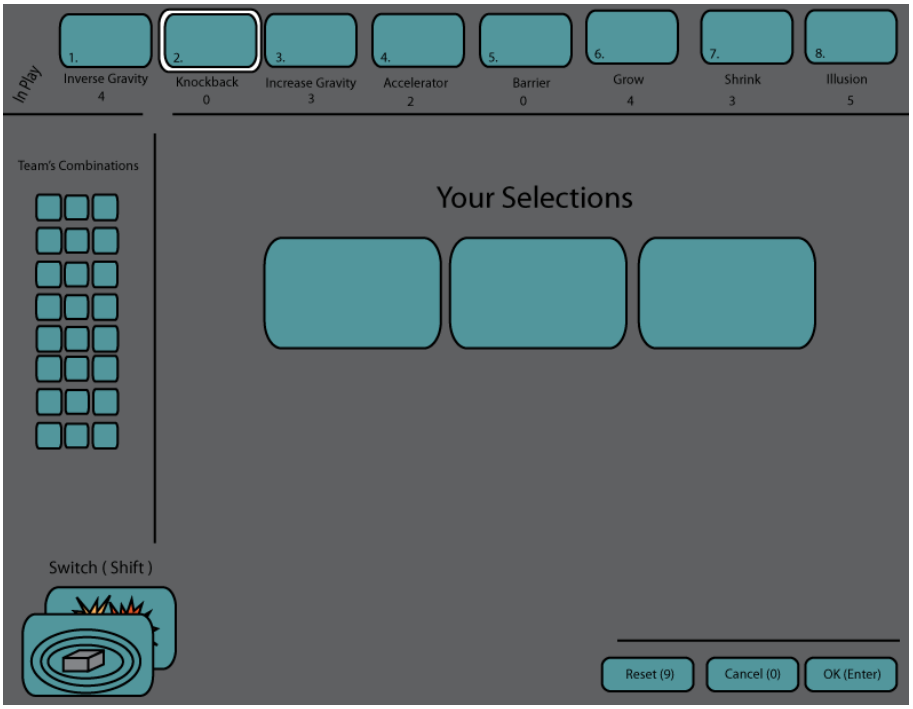


Figure 23: Modifier Selection Screen Mock-Up

Along the bottom right side of the screen are the same three reset, cancel, and accept buttons that are found on the weapon screen. Pressing the reset button resets the modifier combination that the player had chosen at that point. Hitting the cancel button brings the player back to the spawn state unless a game has just started in which case the cancel button is grayed out. The accept button brings players to the spawn state when they are ready to proceed.

In-Game

While inside the game, players have two possible scenarios that could lead them out of the game state. The first is the player is allowed to exit from the match and return to the main menu. This action completely removes the player from the game and they cannot rejoin the game once they have left. The second is when the player dies, they move to the respawn state. This transition is only temporary since they can return once the respawn timer is up. While in the game state, players have important interface elements at their disposal to help them understand their status within the game world.

Shield Meter

The shield meter is the user interface element that players will use to keep track of their shield status. The meter will be located in the top right corner of the screen while in the in-game state. The meter will drop when a player is damaged and will flash red when they are on their last hit before they could die. The red flash will be accompanied by an audio cue to help indicate desperateness. The meter will increase over time when a player is regenerating their shield outside of battle.

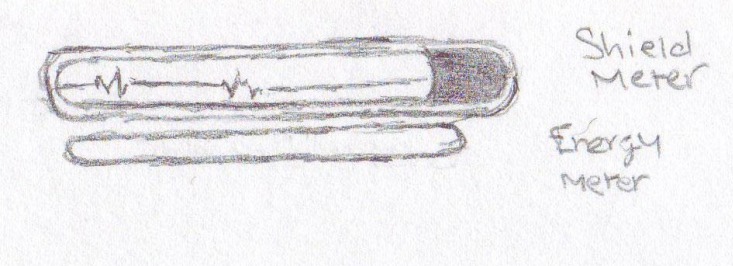


Figure 24: Shield and Modifier Energy Level Concepts

Modifier Energy Level

The modifier energy level bar will keep track of the player's energy resource that is linked to deploying the bar is a horizontal bar that is placed directly under the shield meter. The meter will decrease from right to left, similar to the shield meter. When a player has exerted all their energy resource, the meter will be completely empty and hollow.

Main Target Reticle

While in a match, players have a target reticule in the middle of their screen. This reticle represents the line of sight that the player's weapon or modifier has. The reticule remains white while players roam around the level. When the reticule intersects with an enemy or enemy modifier, it changes to a red color. This helps players identify targets of interest quickly.

Weapon Ammo Meter

The ammo status meter will be located in the bottom right corner of the screen. The meter will display the max ammo the weapon has left as well as how many rounds are left in the current clip.

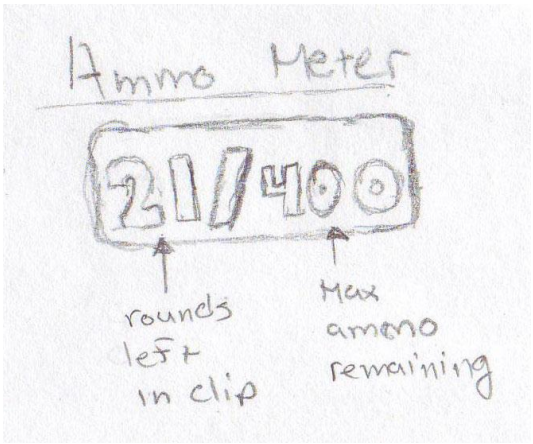


Figure 25: Ammo Meter Concept

Modifier Target reticle

When Modifiers are deployed, they have a target reticle linked to them so players know where to find them and trigger them at later points. Along with the target reticle, an icon indicating the Modifier's type will be displayed next to it at all times. The location of these reticles varies based on the player's positional relation to the Modifier. Players will only be able to see this reticle if they are facing the direction they deployed the Modifier in. If they are facing away from the Modifier, an arrow will follow along the border of the screen until they are facing the Modifier. When the proximity alarm

triggers, the target reticle will flash red. The reticle will continue to flash until either the Modifier is triggered or players are out of range of the Modifier.

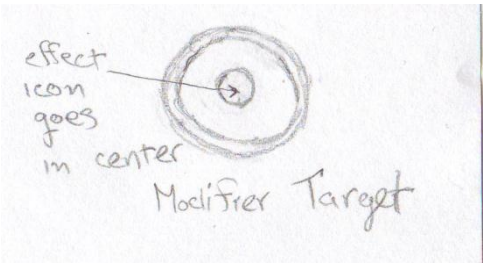


Figure 26: Modifier Target Reticle

Modifier Charge Level

The charge level meter only comes into play when a player is attempting to charge up the modifier they are about to deploy. This meter will appear directly under the target cross hair that remains in the center of the player's screen at all times. The meter will flow from left to right and will be segmented into 3 seconds signifying the three tiers of the modifier. As the meter fills, once it passes one.

Figure 27: Modifier Charge Meter Concept

King of the Hill Interface

It is important for players to know how the tide of the battle is going at any given point in the match. The King of the Hill interface element will be a scoreboard that shows both teams' score. The score is indicative of how many Hill captures each team has achieved. The interface will be placed in the top middle portion of the screen so it is not at the player's immediate attention but follows the same interface line as the shield and modifier energy level interfaces.

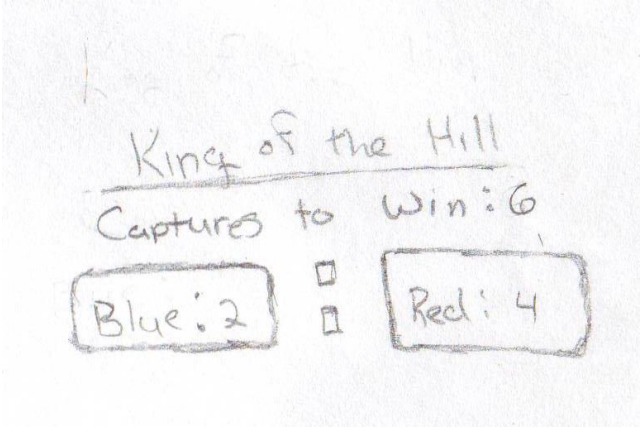


Figure 28: King of the Hill HUD Concept

Assault Interface

The Assault interface will achieve the same responsibilities as the King of the Hill interface in that it will give players an indication of the match status. The interface for the Assault game mode will be a linear graph that represents the attacking team's

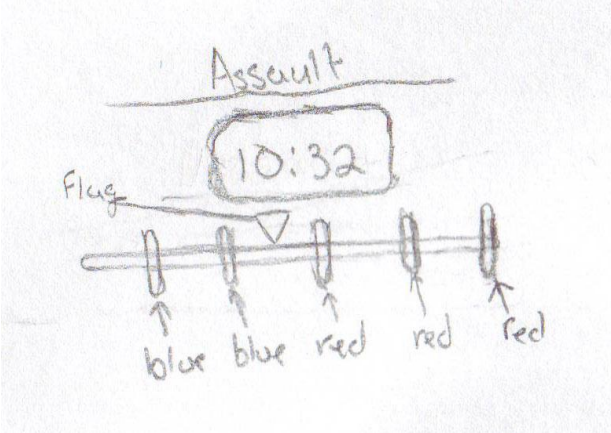


Figure 29: Assault HUD Concept

progress. This progress is dictated on the flag's location in relation to the next checkpoint. The checkpoint will be distinguished by large icons along the graph. When / if the attacking team crosses the required checkpoint's threshold, the interface element for that checkpoint will change from the defending team's color to the attacking team's color. Also in this grouping of interface elements will be the match timer showing how much time is left in the match.

Spawning

The spawning state will come into play at two different times during a game. The first is at the beginning of the game when players are setting up their arsenals. Players enter this state when they have finished setting up their weapons and modifiers and are waiting for the setup timer to reach zero. The second time is when a player dies during battle. At this time, players are brought to a screen that shows the progress of the match in real time. For King of the Hill matches, the screen shows the score of both teams as well as any current capture progress. For Assault matches, a linear flow graph of the flag in relation to the capture points along the level as well as the time remaining in the match is displayed.

In the bottom right of the screen are two buttons: Arsenal Selection and Respawn. The Arsenal Selection button brings players to the weapon selection screen. The Respawn button allows players to respawn when they are ready. This only can happen when the initial 10-second cool-down has finished. Until then, the Respawn button is grayed out and unavailable for player input.

Credits

The credits screen will be broken up into three stages:

- **Development Team**: The students that made the game will be listed in alphabetical order. Each student will their name, a picture, primary responsibilities to the game, and research
- **Assisting Faculty and Applications:** The second screen will give credit the faculty at R.I.T. who were directly involved with development, give credit to R.I.T. itself, and an icon list of applications and tools used to help build the project.
- **Extra Credits:** The third state will give credit to third party sources that helped build additional assets for the project.
- **Team and Contact Information:** The final screen will show the development team's logo, the contact information of the team members, and a link to the *Trigger Happy* website.

All transitions between the various credit states will be a fade to black transition. The transition after the final credits screen will either bring the player back to the menu screen if they came from the menu screen directly or it will exit the game if they are exiting the game.

Look and Feel

Trigger Happy takes place in an absurd, bizarre, and comical Universe. The visual styling is meant to enable, encourage, and amplify these choices. The game's color palette focuses on bright, primary colors that capture the viewer's eye. They also create a positive, happy feeling that matches the world. The rendering style is simple and iconic, focusing on recognizability over realism. Instead of wowing the player with ultra-high quality representations of boring environments, we will devote our time to interesting, meaningful environments that capture the player's imagination. In terms of technology, we will be using a flat shader to achieve our simplified look. However, it is not enough to just flat-shade

normal models and call it a day. The model design must be amplified to communicate its point quickly and distinctively. Not only is this important for creating a unique look, it is critical in styling something as fast paced as an FPS. Emphasizing the key parts of the environment through use of distinctive size, shapes, textures and colors is critical to our look.

**Consult Art Bible for specific details about the environment and game objects.*

Camera

The camera that is linked to players will be a traditional first person camera. The view from the camera will be from the eye level of what the character's eye level would be during a game. The movement of the camera is very rigid as it maps directly to how the player moves their character in the game. While players will not be able to see the model of the character they are controlling, they will be able to see the weapon they are currently firing and various Heads-Up-Display elements.

Audio

To help maintain the feel of the game, the audio will again maintain the concept of an absurd, bizarre, and comical universe. 3D audio will be used for sound effects, as it will provide positional data for players and tell them about things that are not in their field of view. Players will have the option to scale the volume level individually for sound effects and music. Voiceovers will be considered "sound effects" for that scale.

The background music will utilize a selection of electronic instruments and the occasional electric guitar. Music will be composed with the concept of battle in mind, and will help the player to stay "in the zone" in the heat of battle. The music will change based upon the situation the player is in, such as whether or not the player has a low amount of health or a battle is going on nearby.

Sound effects will be developed with the juxtaposition of this world and the killing within, and will be processed to remove some realism and amplify certain parts of the sound. Each weapon and modifier will have its own unique sound for identification purposes. Weapons and modifiers will play sounds when the player switches to them and when they are fired or placed. Modifiers will also play a warning sound when a player is close proximity and a different sound when the modifier goes off. In addition, sound effects will be played when a player lands on the ground from the air or when an explosion occurs. Finally, voices will utilize voice actors and a selection of phrases will be collected. In addition, sounds of human exertion will be recorded in for when the player is injured or lands from a large fall.

**Consult Audio Design Document for specific details on audio assets and implementations.*

Control

The control scheme for *Trigger Happy* has been carefully thought out to maximize the user's experience. Breaking down the scheme, players will find that the user interface is intuitive and direct. Additionally, the controls that players will use a traditional first person shooter style control scheme.

Logical Control

When put all together, the user interface elements during a game will present the player with an understanding of how they and their team are faring during the match. Players will be able to draw information from these interface pieces and be able to create a strategy that will help both themselves and their team.

Physical Control

There are three different control states the player can find themselves in at any point during the game; Menu Navigation, Weapon and Modifier Selection, and In-Game. Each of these states performs distinct functions that directly influence the state the player is in. Each state includes mouse and keyboard functionality.

Menu Navigation

Navigating all menus up until the player reaches the in-game state will be simple and intuitive. The arrow keys allow for quick navigation between menu items as well as utilizing the mouse for selecting any menu item users want, regardless of item order.

Weapon and Modifier Selection

The Weapon and Modifier selection screens will share the same keyboard and mouse input scheme. The number line across the top of the keyboard serves as hotkeys for selecting the various Weapons and Modifiers at players' disposal. Also at the end of the number line are the reset and cancel buttons. Players can also utilize the mouse to select the combinations of Weapons / Modifiers they wish to use. The enter key accepts the player's selections and moves them to the next state. At this point in the game's progression, players will have their hands at the ready on the WASD keys so it is important to make sure players hands do not deviate from this keyboard position too often. That being said, the toggle to switch between the Weapons and Modifiers selection screens are the CTRL buttons on either end of the keyboard.

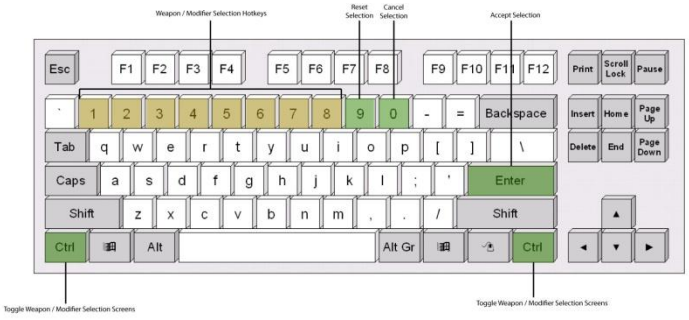
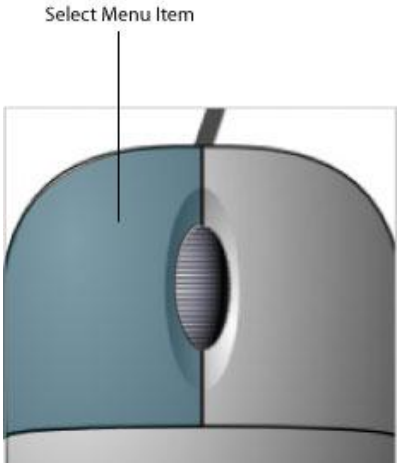


Figure 30: Mouse and Keyboard Layouts for Weapon / Modifier Selection



In-Game

The In-Game state receives the most attention to input scheme as that is where players spend the majority of their time. Players will utilize the WASD keys on their keyboard for movement and mouse movement to turn and aim their character. Players will also have at their disposal the number keys along the top number line as hotkeys for quick weapon selection. Because combinations of weapons can vary, keys 1-4 are dedicated for weapons only and 5-7 are dedicated for Modifiers. This ensures

that players will not lose track of which Modifier is mapped to which key as the size of their weapon combination changes. A secondary mode of weapon selection uses the scroll wheel found on the mouse. The right-click button on the mouse will be dedicated to triggering Modifiers.

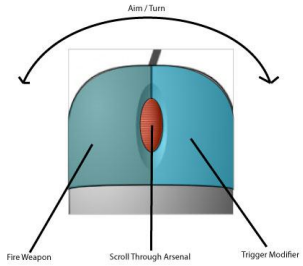


Figure 31: In Game Keyboard and Mouse mappings

World

Historical Overview

The world of *Trigger Happy* is positioned some centuries into our future. The time leading from our present day to the game's present day breaks down into four major periods: expansion, the birth of cloning, the death of cloning, and the proliferation of cloning.

Expansion

One century beyond our present day, Mankind was forced to expand into the solar system due to the increasing overpopulation of Earth. After some fits and starts, Man succeeded in inhabiting a few nearby planets, which relieved a lot of the strain on Earth's resources and improved life for settlers and home worlders alike. In fact, it was so successful that it set off a rush to capture the often scanty (though sometimes vast) resources of other nearby planets. This expansion initiated a boom in the (now inter-planetary) economy that led to the creation of a number of harvesting companies. However, within a century of exploration many companies had proceeded so far outside of the solar system that the only nearby colonies were extremely underpopulated and unable to support further progress (it was hard just finding a place to eat). Of course, the reasonable answer to this problem would have been to wait for populations to increase and to enjoy the abundance of natural resources in the meantime. However, in the calculus of business, the infinite expanse of the Universe plus a burgeoning market for luxury items equaled "find any way to keep going."

For some companies, this meant conducting subtle warfare to capture competitors' resources. However, other companies decided to dump vast amounts of money into R+D to solve the supply chain and worker supply problems. Ironically, the solution was to increase the human population, which had triggered Mankind's problems in the first place. With a combination of industry pressure, vast amounts of money, and virtually no ethical oversight, the first cloning technology was born.

The Birth of Cloning

No one is sure of the exact circumstances, but many believe that the first full-grown human cloning technology was probably created in a dark, dungeon-like lab under the cover of storm clouds. The only thing we do know is that some companies were pressured to put up a memorial to the "courageous"

test subjects lost in the process of "discovery." While at first a trade secret, the extremely expensive cloning process soon spread around the galaxy because of corporate espionage. Corporations with the banked money to make the bottom-line work out in their favor spawned outposts composed of singled-minded workers that enabled them to press farther on. Unfortunately, expansion only lasted only a few decades more at which point the long expected (and denied) edge of the galaxy was discovered. At this point, the shaky peace between corporations began to breakdown as the full resource map was now known and it was finally possible to know who was ahead. War became the only method of increasing market share. The situation appeared to revert to the same as the initial boom, but many forget that clones could be used for more than mining and shipping.

The Death of Cloning

As corporations declared war on each other, it shortly became clear that the battles were not going to end soon, if ever. Based on the requirements of the cloning process, existing resources, and the attrition rate of combat, it was calculated that war between the two largest corporations would conclude in roughly 3.1x10^7 years. Recognizing the futility of their conflict (but only after harvesting at least 50% of all the galaxy's resources), the various large companies declared peace and set out to undermine each other in different ways. Within a century, the market had become saturated with goods, and the only people that could afford large scale cloning began to fade away.

The Proliferation of Cloning

As science caught up to greed, the cloning process was refined and cheapened. Alternate methods were also developed, and the price eventually dropped to the point that a normal person could afford one or two backups if they saved for them. Shortly after, employers began to demand that potential employees provide proof of clone to offset any potential losses resulting from death. This, more than anything else, spread clones to the various different pockets on humanity (some of which were originally clone colonies themselves). It became commonplace to know at least one person than had been "swapped" (replaced with a clone) over the course of their life (though fidelity in cloning methods varies and some had woken up one day to find that their swapped significant other was borderline unrecognizable).

In the present day, clone prices have declined enough for most people to support a small group of high fidelity clones. For those with fortune or sponsorship, a vast supply is not beyond reach. More commonly, people buy "upgrades" to their current bodies in the form of trendy new parts. Everything from pragmatic benefit (more strength, dexterity, damage resistance) to cutting edge fashion (glowing, patterned) is available. Unfortunately, after centuries of wide spread cloning, the corpse waste from the past wars and clone consumerism has started causing environmental issues.

Life in the Modern Day

Habitation

Modern life is largely terrestrial in nature. Even though technology has advanced far enough for humans to live in space, the large majority prefer to live on some planet or other. Travel in space remains expensive and/or dangerous at a personal level, so trips are scheduled in way akin to our current airlines. For this reason, most work within their planetary ecosystem unless they happen to be involved in some form of intergalactic business. The exact living conditions vary from planet to planet, but most have a number of terrestrial cities in the most habitable areas. Particularly rich or advantageously located planets can be very populous, but most are less dense than the Earth today.

Planets nearer to the Earth are more likely to be populated, but are also more likely to host ruins from our near future (ghost planets).

Modern Cloning Technologies

A number of cloning methods exist in the modern era. Some are refinements of the original method, while others are new. They each have their own strengths and weaknesses.

- **Build-a-Body** – The most popular cloning method employing standardized, interchangeable appendages to reduce clone construction cost. Customers are measured to determine which parts best match their own natural form and are given a full specification that they can later use to order a supply of clones. Alternately, customers can choose "designer" parts to replace their natural ones (excepting the face, which is illegal to modify for reasons of identification). Build-a-Body suppliers frequently run promotions featuring new parts, which creates a consumer culture of constant upgrades and outdated models. Customers can switch between their purchased bodies using locally available brain transplant centers (thanks to the ISO-9000 Brain-Body Interface standard and frequent brain backups).
- **Authentic Cloning** – A more expensive cloning method based the original process. It works at the genetic level, allowing the maximum customization possible, but requires a longer, more resource intensive gestation period. Only the rich, vain, or people that *must* be different need apply.
- **Multi-Cloning** – The most expensive method of cloning by far. It involves the upkeep of a full set of clones from the customer's birth. Usually, an allotment of 10-20 clones will be purchased and then raised in various different locations and living conditions. This scheme has the advantage of allowing the customer to explore many life paths simultaneously and to choose the most pleasing one. Clones following rejected paths are either allowed to live free or terminated.
- **Ancestral** – Often used in situations where cloning technology is shared amongst a village or a particularly important body is vacated. It involves a one-time fee paid to transfer a (usually natural born) person into the body of a recently deceased one. It is so named because it is mainly used by tribal cultures to place the mind of a new leader into the inherited body of the old one. Why they choose to keep these bodies around for generations is unknown, but most think that it is for symbolic, religious, or nostalgic reasons.

Organizations

There are two main groups in the FPSA: the Appendage Conservation Front (ACF) and the Impossible Possibilities (IP). The ACF is an environmental conservation group formed in response to the resources wasted by clones. The ACF pushes recycling of clones and condemns the waste of clone resources, and are very similar to our world's recycling advocates. They push the FPSA to allow students access to recycled or refurbished body parts in the event of an injury rather than a full-on clone replacement. The IP, on the other hand, is a group that pioneers new technology in the world of *Trigger Happy*. While maintaining a peaceful front, this group secretly trades the forefront of technology to the FPSA for the ability to use students as test subjects for new experience.

The Industry

Ever since the advent of reasonably priced clones, there have been jokers and daredevils that have exploited the entertainment opportunities inherent. It is from these brave pioneers (mostly using viral

media to distribute their work) that the "industry" as we know it was born. Hits like *Last Safari*, *Candid Catastrophe**, and *Real World Asteroids* propelled them into the public eye and earned them the first network contracts for this type of entertainment. As clone programs (which they came be to known as) began to multiply, genres were identified and explored. The genre that was most important for our industry was originally called "historical reenactment." This class of programs (and later on events) took their inspiration from past battles and utilized the talents of the few warriors that still studied combat. A single "re-nact" pitted teams representing each of side of a historical conflict against each other in "realistic" locations. No one is sure if the original re-enactments were ever truly accurate in a historical sense, but as their audience exploded them intentionally or unintentionally bent the rules more and more (see the "telephone theory of history"). For example, *Spartans XVII: Xerxes Space Giant* was not much like the battle as scholars know it, but it did set a record for simultaneous viewers.

A natural result of the continued rule bending was the expansion of re-enactment programming to include many non-historical combat scenarios. The first example of this type of program was *Rockets vs. 100,* a game show where 100 randomly chosen participants are provided with clones and tasked with taking down a single contestant with a rocket launcher. Needless to say, it was both pioneering and a breakthrough hit. Running simultaneously with the diversification trend was its emergence from the video programming studios. Real world competitions began to draw as well as their video counterparts (even in 3D segments). However, the real breakthrough was the personalization of combat.

The joint forces of cheapening clones and audience expansion meant that investors could afford to sponsor the cloning requirements for a small team or single participant. Unlike the industry stars, these teams could cater to smaller clients and even personal requests. It is at this time that we witnessed the birth of personalized performances for large parties, conflict resolution, commercials and even pranks.

The remaining time between then and the present was largely dominated by the standardization of the industry. Leagues sprung up for hopefuls looking to secure a rare job; they usually stuck around until their supplies of clones ran out (and sometimes, tragically, even afterwards). However, making it big is still nothing but a dream for most. However, if one is willing scrap, there is an alternate path to stardom that involves using low paying, oddball performances to jump-start a team. It is this path that the Future Proficient Soldier Academy initially takes.

**A particularly gruesome show that used hidden cameras to film public reactions to clones being killed in various staged ways. Almost resulted in the banning of all clone programming.*

Future Proficient Soldier Academy

The FPSA was founded three years before the present day. It is the first school to offer instruction in the various fields associated with clone-based combat. It runs its own school ("The Academy"), and maintains partnerships with two other organizations that offer their own services.

The Academy a.k.a. "Trigger Happy High"

The Academy teaches courses in two main areas. The first covers combat skills and the performing arts required for being a professional combatant. After a baseline of entertainment prowess is established, students are taught a variety of weapon and tactical skills. Finally, they are given the opportunity to develop a specialty and a persona to sell themselves. It also features minors in various types of historical warfare. The second area is an art program focusing on crafting weapons and modifications. Students select a specific class of weapon or modification as their focus and produce a

working example as a final project. Admission to the academy is based on a portfolio for craft students and a skills assessment for combat students.

Appendage Conservation Front a.k.a. "Spleen Peace"

The ACF is an environmental conservation group that is part of a larger network of volunteer environmental organizations*. It plays a role in several areas, but its association with FPSA is through its part recycling services. The ACF was formed in the past as a reaction to the corporate wars. They objected not so much to the violence as to the massive waste of resources that it caused. In their eyes, perfectly good body parts were being discarded and replaced with full clones for no good reason. In response, they pioneered the first clone recycling methods, which both created a low cost, clean market for appendages and began to deal with the pollution.

Centuries after their creation, environmental damage from the past wars still exists along with the new scourge of clone consumerism (resulting from cheaper prices). Their current mission is to advocate for restraint in purchasing new clones and encourage recycling of parts. This mission has a few parts. One is their hospital wing, which deals with the emergency care -- refurbishing discarded limbs for use in repairing bodies instead of replacing them (how quaint). FPSA began working with ACF as a way to cut costs on clones for their demonstrations (read: make it possible to operate at all). They tolerate the ACF's "enthusiasm" for their cause because they have no other choice. This sometimes even means participating in ACF "diplomatic missions" as support crew. However, the students are given access to cheap body part replacement (a painless procedure, but resulting in some mismatches because this ACF branch gets most of the poor condition models) if they cannot afford their own clones. The other half of this branch's mission is to popularize re-purposed parts among the general populace. Their plan involves jump-starting an underground fashion movement centering on matching up fashions from the past. The Anti-Common Fashion (ACF) line is the heart of this movement and draws on the best of their recycled parts.

**Not to be confused with the Alien Conservation Fund, which collects money to fight for alien rights in the case that we ever find any.*

Impossible Possibilities

Impossible Possibilities is a research and development lab that maintains a branch in the FPSA building. They specialize in developing technology for the clone combat industry and operate as part of a powerful media corporation. Their official agreement with FPSA is that they provide some materials and weapons to students and in exchange get first shot at to recruiting. What is actually happening is a different story. The Impossible Possibilities corporate headquarters has little respect for FPSA programs and mostly maintains a presence there to use students as cheap test subjects. They also use it as an easy way to paint their monolithic corporation as a community player. As such, they do not commit a whole lot of resources to the operation and only check up on a quarterly basis. Despite what HQ intends, the lab is run as a shell that lies to corporate while screwing around with the provided or found materials. It is largely staffed by Impossible Possibilities employees that are being punished for some meaningless transgression and therefore are not too keen on following the company "vision." Whenever it comes time for a review, they clean up the random experiments and invent a new set of fabricated claims to prove that they are making progress. The best of the crafting students can often be found in the lab working on something dangerous or recruiting their combat friends to test it out.

The Society for Historical Beatdowns

The Society for Historical Beatdowns (SfHB) is an academic institute within the Academy. Its students and professors do not practice combat professionally, instead opting to study its history. Graduates go on to become event planners (especially in re-enactments), consultants to tech designers, and occasionally historians. The range of their study extends back to ancient times, though a lot of the evidence from even recently is lost (due to overpopulation and loss of knowledge). Consequently, they have to fill in gaps with video, books, and sources of uncertain merit that happened to survive. A lot more imagination is used in their exploration of history than was previously the case. The department has recently embarked upon a project to produce semester war memorials that recreate an event in exacting detail (from their point of view anyway). Their base of operations is a museum displaying the various historical pieces that they have bought or found. The museum tries very hard to look professional, but the SfHB does not have an unlimited amount of money so there is some trickery involved (they do have more money than the other three departments, but a lot is spent on the historical expeditions and maintaining their expansive collection). They frequently bring people from the industry though their museum because in addition to research, they act as consultants for big name productions.

Characters

Appendage Conservation Front Staff

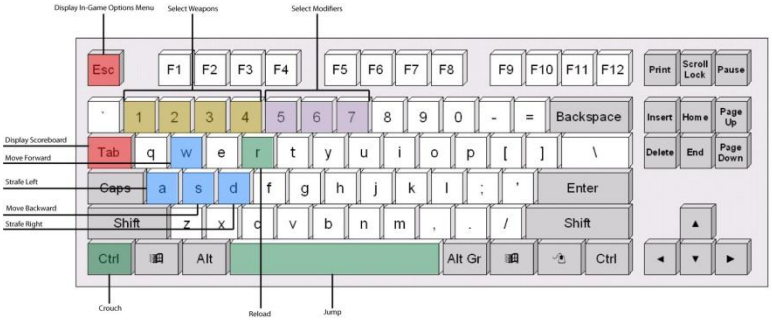
The ACF staff is composed partly of volunteers and partly of full time staff. They vary in their levels of zeal for the environment, but most are on the dedicated side given the sometimes grisly nature of their work (though some of them are in it for that part). They have varying opinions about their assignment to FPSA. Some believe that supporting an industry that uses a tremendous amount of clones only for entertainment is not worthwhile. Others take the more practical approach that they get to make the most impact here while popularizing their program and also getting free rent. For the former group, the FPSA hospital also acts as part of a larger distribution network that provides parts to the disadvantaged, which satisfies their world saving desires.

Personality wise, the group tends toward the eccentric side. They approach their job with a cheeriness that some might find to be a bit creepy. At the same time, they have a nurturing instinct that can surprise you. Some are more interested in the helping side than the processing side (and have been known to get sick when walking in the wrong door). Others are more determined in their approach, hoping to use volunteer work to propel them into a good med school. If you get on the wrong side of the ACF, though, you can expect a harsh but creative condemnation of your position*.

**In the past, they hacked a designer cloning machine to create a batch of hands that would extend a certain finger on random occasions.*

Impossible Possibilities Staff

The IP staff is composed of scientists that have become disfavored in their corporation for various reasons and "promoted" to the FPSA lab. Some of them were too interested in a topic of no commercial value, some had a disagreement with management, and some were a bit unstable. All of them are now less than interested in conforming with their corporate headquarters (except for during quarterly checkups). Initially, they were discouraged about being assigned to an underfunded lab, but they have



come to love the students, the freedom, and their newfound "rebellious" side. Some maintain big dreams about pioneering new fields, while others just want to blow stuff up.

Society for Historical Beatdowns Professors

These professors are more academic than their counterparts. They focus on education as an end instead of as a tool to become better at a trade or craft. Their particular interest is the study of war history and they are recognized experts in the field. They know this, and consequently affect a pomposity that can make them hard to deal with. This arrogance is compounded by the fact that the industry frequently consults their experience and compensates them well. Of course, they are completely unaware that their interpretation of history is wildly inaccurate, but no one else knows enough to challenge them. Many of them were relatively unknown professors before, and some prefer a more academic/conservative style. Others, however, have been struck by the fame and fortune, which creates some tension with their colleagues.

Students (Player Avatars)

The player takes on the role of an FPSA student. The students are humanoid and come from various backgrounds. Some prominent groups include:

- People on scholarship that could not afford to are discovered in the amateur leagues.
- Fans of the industry that show some glimmer of promise.
- Rich students whose parents forced them to attend college but did not bother to examine what the Academy teaches.
- People who will argue to the death that combat is an under-valued but extremely "meaningful" art form.
- Middle aged people on their mid-life crisis.
- The occasional prodigy with a requisite devoted following.

Visually, the students will vary over the course of a battle. As their forms sustain damage, they swap in used parts that may or may not match their previous parts. The change in body parts (excepting the head) is represented by changes in texturing instead of shape. Some examples of modifications are:

- Skin color
- Tattoo
- Bracelets
- Hair
- Clothing

Changes in body parts may also award a temporary bonus with some probability.

Academy Professors

The Academy professors come from a multitude of different backgrounds. Most of them have industry experience of one type of another.

Maps

The Academy

Location Layout

The layout of the Academy can be broken down into various wings. Below is the overall layout of the Academy level. The first image shows the blueprint of the level, which gives the overall shape. The second image is a map of the connections of each of the rooms. The connections map makes it easy to see how each room is laid out. The Academy level on the whole is comprised of 13 rooms. These rooms have multiple connections that allow for a number of different paths that the player can take during the match.

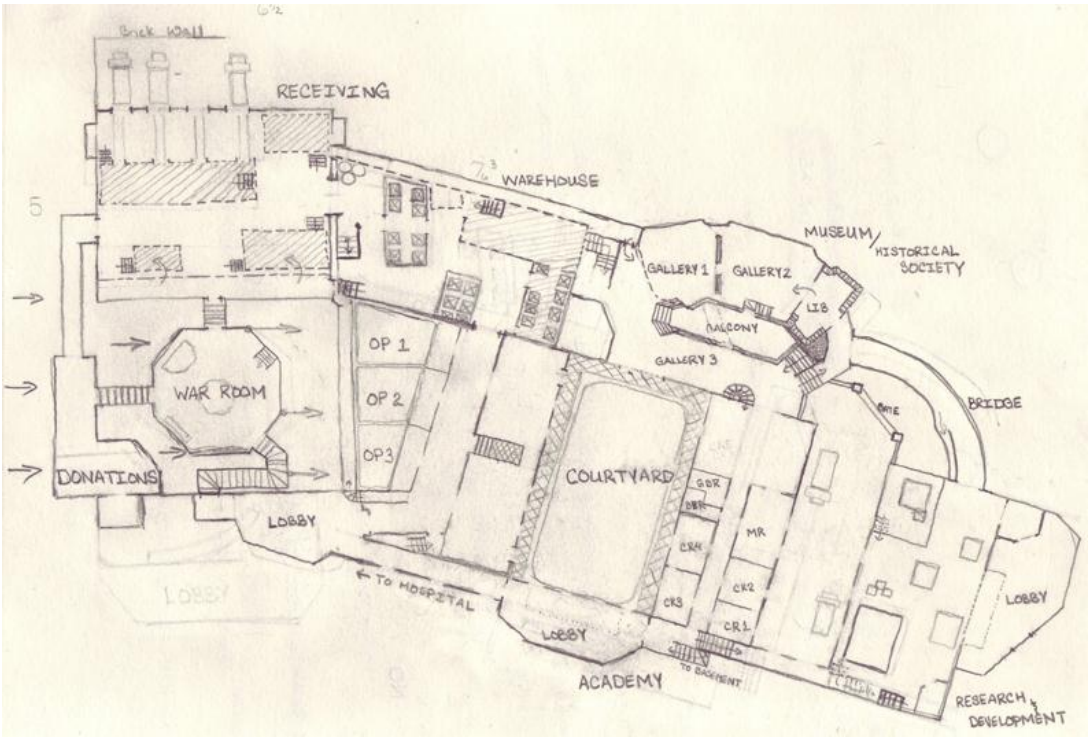


Figure 32: Blueprint of the Academy level

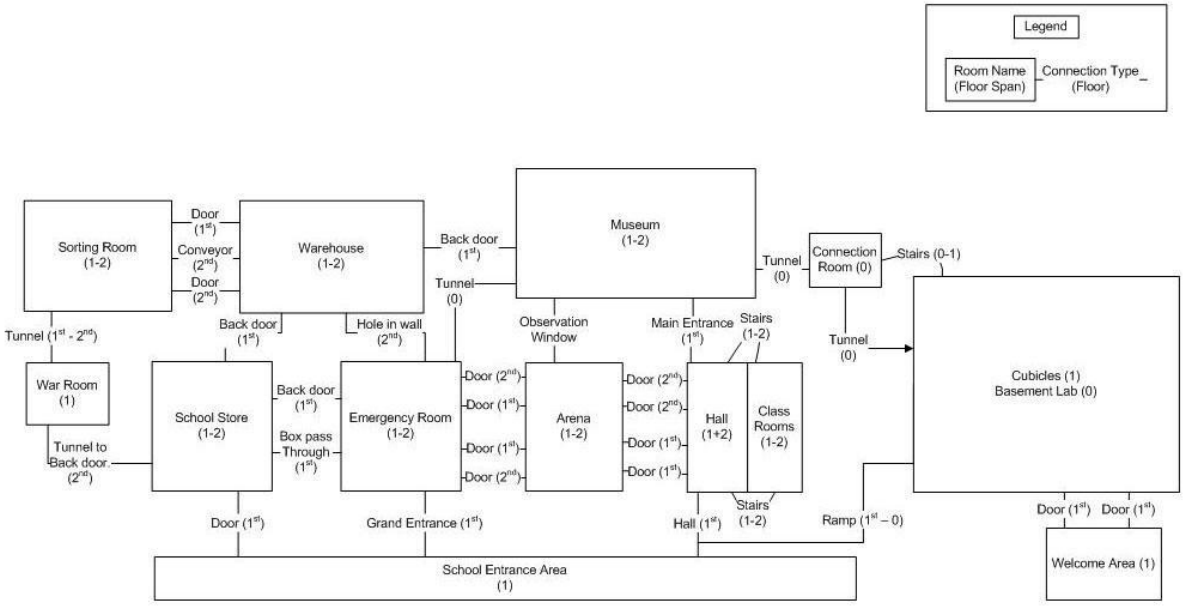


Figure 33: Map of connections for each room in the Academy level

Scenario

At the end of every school year, the ACF and the Impossible Possibilities staff host a competition taking up the whole FPSA building. This competition serves three main purposes. First, it celebrates the graduation of students into the next grade. IP releases its best inventions for one-day use while ACF breaks out the best parts it has managed to collect over the year. Second, it gives the students a chance to showcase their skills and projects before their peers. Parents, friends, and non-combat students gather outside the building to cheer the teams on while watching video feeds. Third, and most importantly, it decides which group will command power in the following year. If ACF wins, they get full access to IP's technology for a week (the last time this happened, they modified a pallet of expensive arm upgrades to make them randomly brandish their middle finger), while if IP wins, they get a couple bins of mannequin parts to test on and amnesty for all of the parts they've stolen over the year.

Location Design

The FPSA Building

The FPSA is housed in a large, solid building that was converted from an old military fortification. It was left over from the wars a thousand years ago like medieval castles in our present day. It was chosen for its space and sturdiness (it has to resist bullets and explosions), but mainly because it was the cheapest rent that they could find. The building is stone on the outside, but has crumbled in some places. The broken down parts as well as sections of the interior have been refurnished with wooden or plaster walls on the inside. Some of the rooms even have modern systems for providing heating and other services. Compare this to the surrounding city, which is built from new-age materials that enable artistic, gravity defying architecture.

The FPSA building is two-stories tall, with a bunker/basement. Its rooms are arranged in a number of different configurations due to its military use. Some of them have recently been connected (though

intentional or unintentional demolition) to enable certain layouts for teaching. The building is composed of four main wings: the Appendage Conservation Front wing, the Impossible Possibilities wing, the Academy wing, and the Society for Historical Beatdowns wing.

The Appendage Conservation Front Wing

This wing houses the ACF. It is a combination of a hospital, school store, and distribution center.

- **Sorting Room** – Where the ACF receives parts donations and sorts them. It has a number of bays to accept trucks full of wooden boxes with parts. These bays are connected to conveyor belts that sort the parts into upper body and lower body categories. The sorting conveyors span two vertical levels.
- **Warehouse** – The warehouse stores boxes of parts according to various criteria including gender, size, etc. They are kept in large racks that have been re-purposed from holding weapons to hold various boxes. Above the doors are signs that warn players to avoid creating sparks of fire in the Armory (which was the original function of this room).
- **War Room** – A hidden room where the ACF plans their "missions to promote awareness." It contains maps marked with objectives, ammunition, and a central discussion table.
- **School Store** – Where students come to buy nicer refurbished parts and school supplies. It is broken into sections by school function, with each section having a number of "dressing rooms", boxes of parts, and clothing to match the parts. Ads cover the walls promoting the different styles (most of them "retro"), the school, and the environmental benefits of buying used.
- **Emergency Room** – A room designed to repair students in the fastest manner possible. It has gurneys on rails that convey injured students through a large machine that pulls random parts from a hopper and heals them. It is located below the school store.

The Academy Wing

This wing contains the classrooms and training areas used primarily by combat students (and sometimes crafting students).

- **The Higher Learning Arena** – A three-level arena constructed to simulate a variety of combat situations. It is marked up with large decals that identify important spots (learning opportunities) including sniping spots, beneficial mod locations, and spawns to camp.
- **The Classrooms** – A two-level network of rooms used to teach various subjects. They are connected by a complex series of pass-throughs that were put in place to reshape the original rooms into a form better suited for the classes.
- **The Classroom Hallway** – A long hallway that contains the entrances to all of the classrooms. It also has the student lockers.

The Society for Historical Beatdowns Wing

This wing is dominated by a large, museum-like set of displays that chronicle the history of warfare.

- **The Museum** – A large area with a number of displays positioned around it. It is better decorated than the rest of the school because industry representatives frequently drop by to consult with the professors on new programs. Example displays include:
    - **The First Exoskeleton (roughly 1845)** – Based on cannons found within the FPSA building and recovered illustrations of steampunk contraptions. The William Stevenson Co. Exoskeleton wielded a cast-iron cannon and a ramrod, and was powered by a steam engine. It is thought to have been reloaded by gunpowder toting assistants.

- **The Bowmerang (ancient)** – Based on a notched, flexible piece of curved material found near a label that said "Bows." Scientists immediately realized that "Bow" is a shortened version of "Boomerang" and hence concluded that it is a thrown weapon used to decapitate enemies and return to the owner. It is believed that a single, serrated cord connects its two notches (though the strand was never found) and is fired by introducing tension into this cord.
- **Immaculate Creator Be Merciful Totem (1900s)** – Based on a large rocket marked "ICBM War-head." The large, obelisk-like form of the ICBM identifies it as a religious object. The word "War-head" names the (likely violent) god that it was dedicated to, while ICBM is an invocation possibly meaning "Immaculate Creator Be Merciful" (asking for the god's blessing). It was worshiped in a ritualistic fashion before an important battle.

Impossible Possibilities Wing

This wing is built to be deceptive. One floor projects an image of corporate professionalism while the other reveals that type of chaotic experimentation that is actually going on.

- **Ground Level** – This level is office-like, featuring modern equipment (floating glass and steel), motivational posters, and a cubicle layout. It is heavily monitored by corporate HQ using a series of cameras, but they still have yet to realize that most desks are staffed by mannequins.
- **Basement Level** – This level features multiple testing areas in various states of destruction. The exotic technology is meant to be quickly hideable in case of a visit from corporate headquarters.

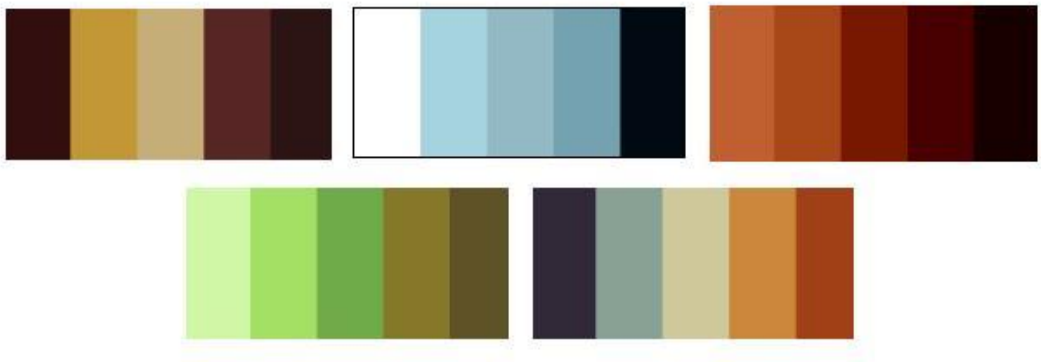
Art Bible

Theme and Style

[Note: All sketches detail shapes and colors, not rendering style. See descriptions and references in Character and World Style sections for that information.]

Visual Style

Trigger Happy takes place in an absurd, whimsical, and comical Universe. The visual styling is meant to enable and amplify these feelings. The game's color palette is diverse, featuring distinctive colors that uniquely mark areas and characters.



The rendering style is simple and iconic, focusing on recognizability over realism. Instead of wowing the player with ultra high quality representations of boring environments, we will devote our time to interesting, meaningful environments that capture the player's imagination. The model design will be focused on communicating its point quickly and distinctively. Not only is this important for creating a unique look, it is critical in styling something as fast paced as an FPS. To do this, we will be emphasizing the key parts of the environment through use of distinctive sizes, shapes, and colors. We will also simplify the texturing and geometry to move the player's focus from the rendering to the objects. Simplification should not be confused with low quality, however. It is a high quality style that uses a small number of colors and details to nonetheless capture the meaning of a texture or form (and even amplify its meaning through amplification by simplification).

Simplified Textures and Geometry (Team Fortress 2)

Complex and Realistic (Crysis 2)

Another reason for choosing a more iconic, cartoonish style is the subject matter and tone of our game. Trigger Happy is designed to create comical and fun interactions. However, it is also a first person shooter featuring body part swapping. A realistic look would create mixed feelings because of these off putting topics. For some games, fully representing the gore adds to the design (such as a gritty war game), but for our fun and light feel it only detracts. Cartoon violence is appropriate because it sidesteps most of the real world consequences and shifts focus back on the inventions that we create for the game.

Character Style

The player avatars in Trigger Happy are first and foremost normal, humanoid students. Second, they are FPS combatants that have little money. Their shape and clothing is a result of these two forces combined. Each team member wears a simple set of armor composed of scrounged parts such as pillows, sports pads, and cooking ware.

The characters' bodies are stylized instead of being anatomically correct. They are not cartoon characters interacting in a cartoon world, but they also are not ultra realistic in the style of Call of Duty or Gears of War. Their forms are abstracted to smooth over the details of body hair, subtle musculature changes, wrinkles, and small joints. Their textures are low frequency, simply detailed, and realistic in color. The character models are lit to emphasize their silhouette, which means brightening the model's border in comparison to the interior. Shadows will be modified to rapidly fall off, creating defined shadow areas on the body.

References

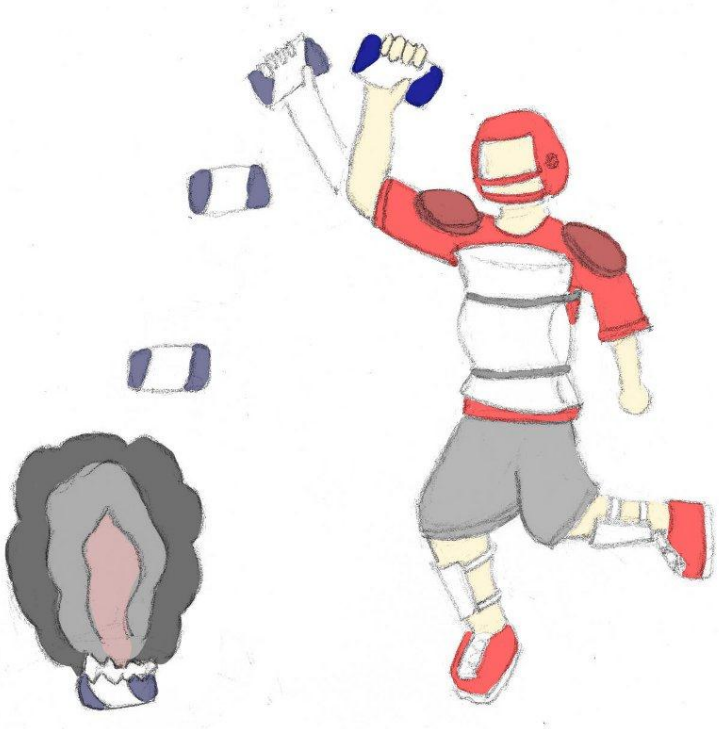
Sniper – Team Fortress 2

Relevant Parts: Distinct border. Simple textures. Abstracted body shape. Slight, cartoonish feel.

Human Female – World of Warcraft

Relevant Parts: Abstracted, idealized body shape. Realistic, but also lower detail, simplified textures.

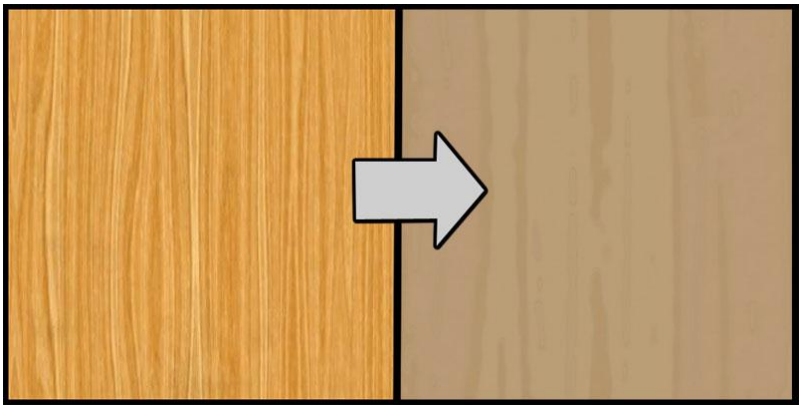
Concept Sketches



World Style

The world of Trigger Happy is designed to mesh with the appearance of the characters and the comical feel of the game. It is represented in a naturalistic style but with much of the detail abstracted away. Textures are simplified, low-noise, featuring broad strokes of color derived from more detailed base textures. Only as much detail as is needed to communicate the object's material is provided. The color choices will reflect the palettes of the areas that the objects reside in. Objects are modeled with low complexity. They are never shiny or reflective, instead using diffuse lighting. They are meant to communicate their purpose quickly and effectively. The general feeling is simple and clean from a visual perspective. Interest is generated though the content of the objects instead of the rendering.

Below is the result of transforming a source texture into a texture matching our style. It involves abstraction, smoothing, and then flattening and desaturating the colors without losing the significant details that define the texture.



References

Warsow

Relevant Parts: Textures that prioritize shape and color over high frequency detail. Simple geometry.
http://www.warsow.net/media/0.5/1280px/01.jpg

Team Fortress 2

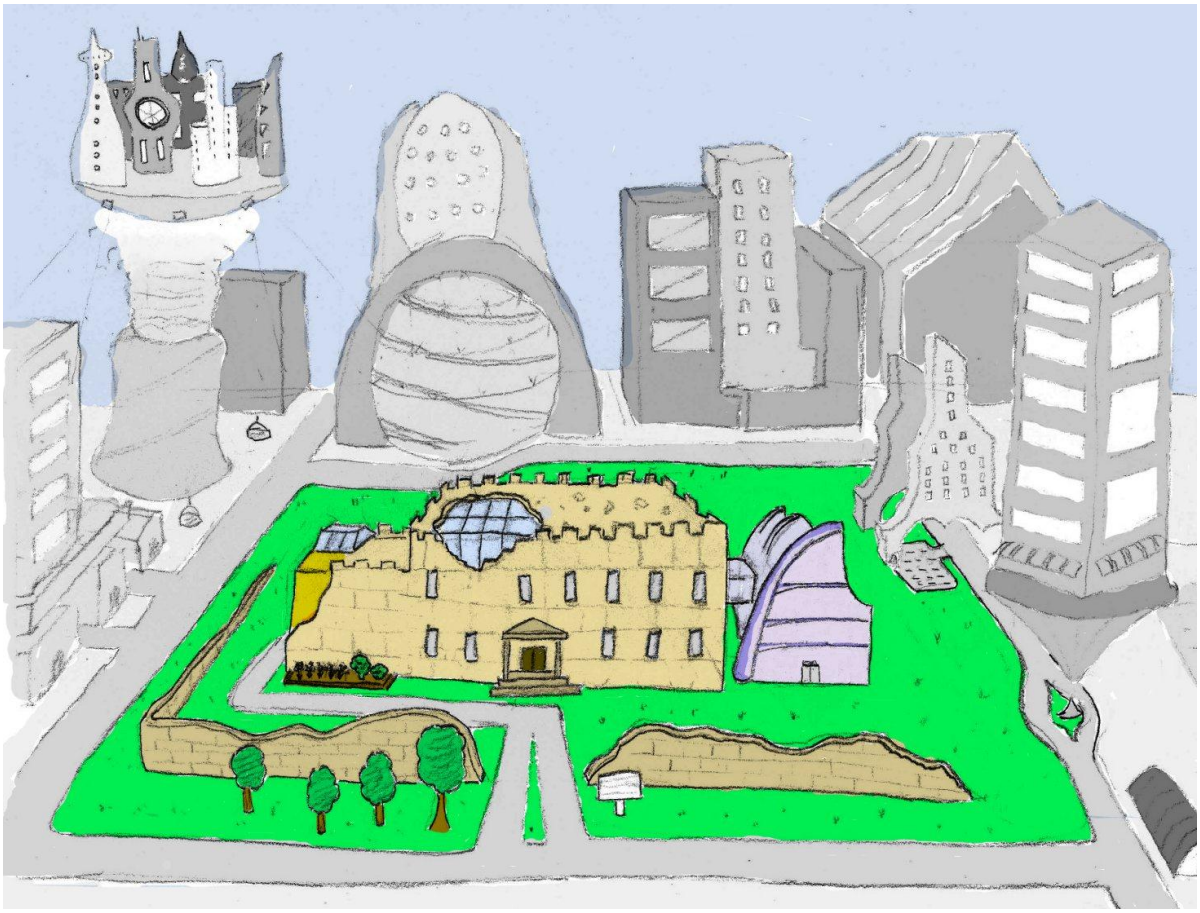
Relevant Parts: The painterly quality of the textures. The angular and orthogonal lines. The eye-catching layout of the geometry.
http://evilwombat.files.wordpress.com/2008/08/05_ravine_2.jpg

Map Styles

The Future Proficient Soldier Academy

The Academy is built into an old, stone fort that has been decaying for several hundred years. It is two stories tall with additional space in a spooky, catacomb-like basement area. Its rooms and corridors are in various states of disrepair, ranging from surface damage, to structural damage, to complete collapse. Some parts have been patched up with new or found materials, but most have not. The original rooms of the fort have mostly been converted over to new purposes for use in the school. For example, the armory now stores spare part boxes as a warehouse, while the dungeons have been re-purposed as a hidden experimentation area.

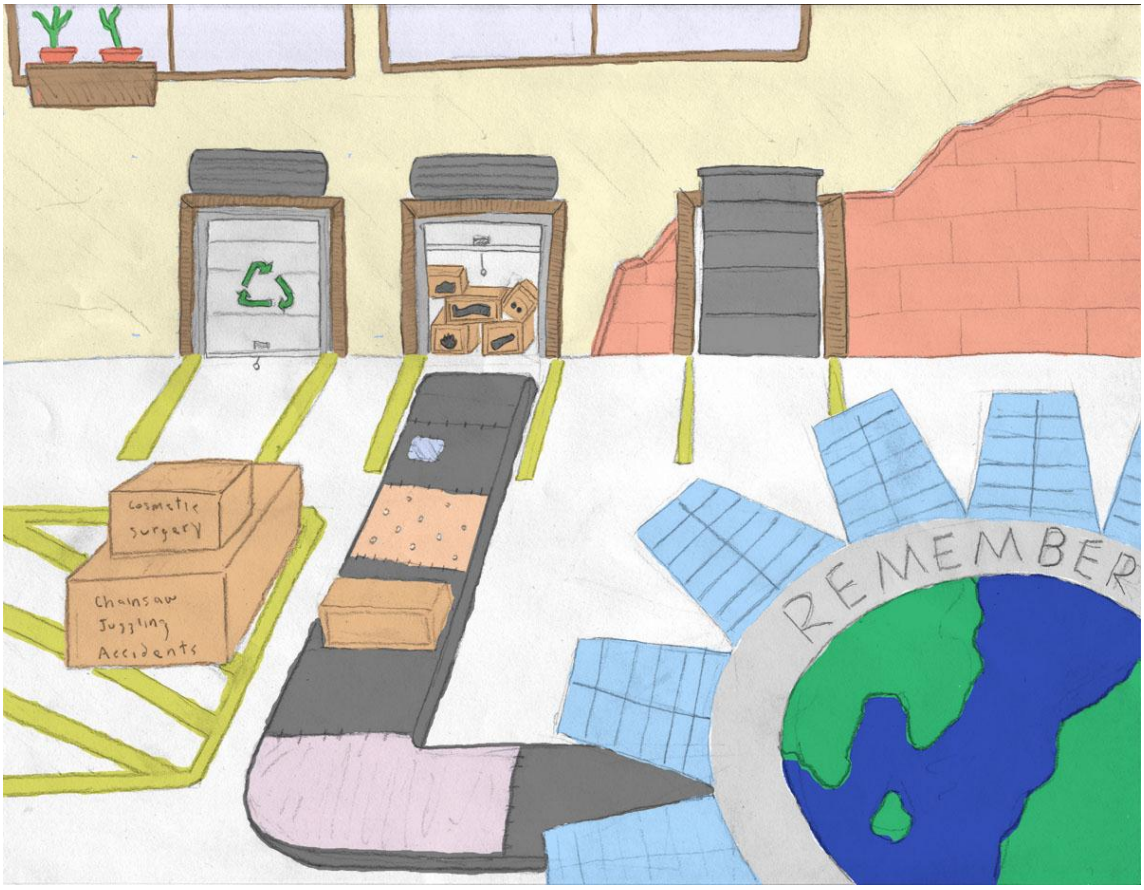
Attached to the fort is a new research lab built by the fabulously rich Impossible Possibilities Corporation. Unlike the rest of the school, it is built from high quality futuristic materials. In fact, the entire city surrounding the Academy is more like the research lab than the fort. The only reason that the fort and its grassy grounds exist in the modern architectural age is that universal historical preservation laws prevent its outright destruction.



References

The Appendage Conservation Front Wing

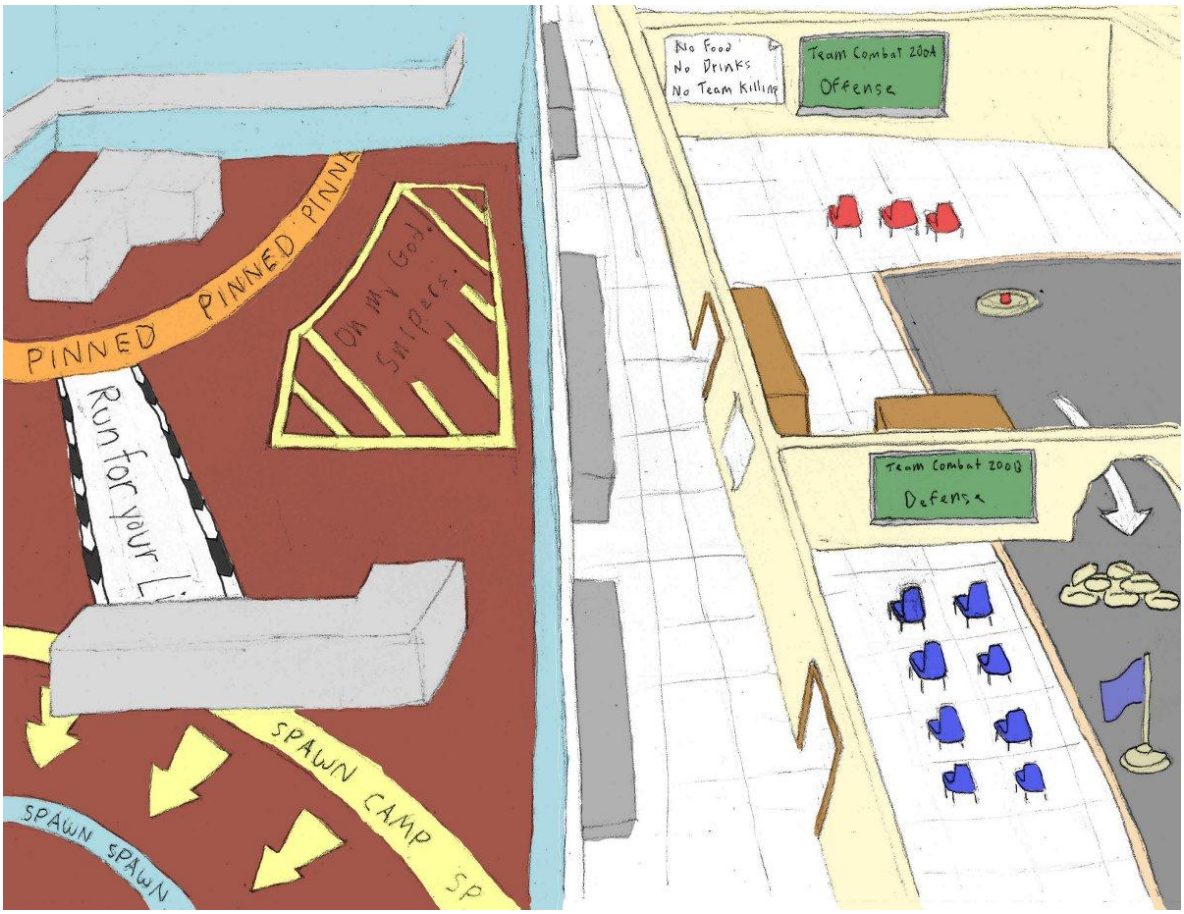
The Appendage Conservation Front Wing is half dedicated to ACF work and half dedicated to their school support activities. It is styled in a bright and cheery way, which is meant to be juxtaposed with its morbid work. It is the wing with the most windows, light, and airy space. The decorations placed throughout the wing are mostly natural or do-it-yourself kind of objects though there is machinery when it cannot be helped. For example, their conveyor belts are mostly metal and rubber, but the holes in them have been repaired with cloth patches that betray hand stitching. The walls are partially fort stone, but it is vine covered in places. There is a prodigious amount of signage advertising environmentally friendly practices. Any repair work done here is done using sustainable materials like wood or clay.



References

The Academy Wing

The Academy Wing is housed in the fort training area. This area is composed of a number of training rooms and a large courtyard that was used for doing military exercises. It is styled in a semi-scholastic fashion, with notable differences that separate it from a standard school look. These differences are largely drawn from a vast store of first-person shooter objects and training items. The point is not so much to mock a FPS, as it is to create an environment where students can learn about this type of combat. As such, every cover spot, spawn location, and capture point is a labeled learning opportunity. The materials here are rubber for the training arena, and stone for the classrooms. The classrooms contain a mix of rugged training gear and standard school gear like chalkboards and class rules.



References

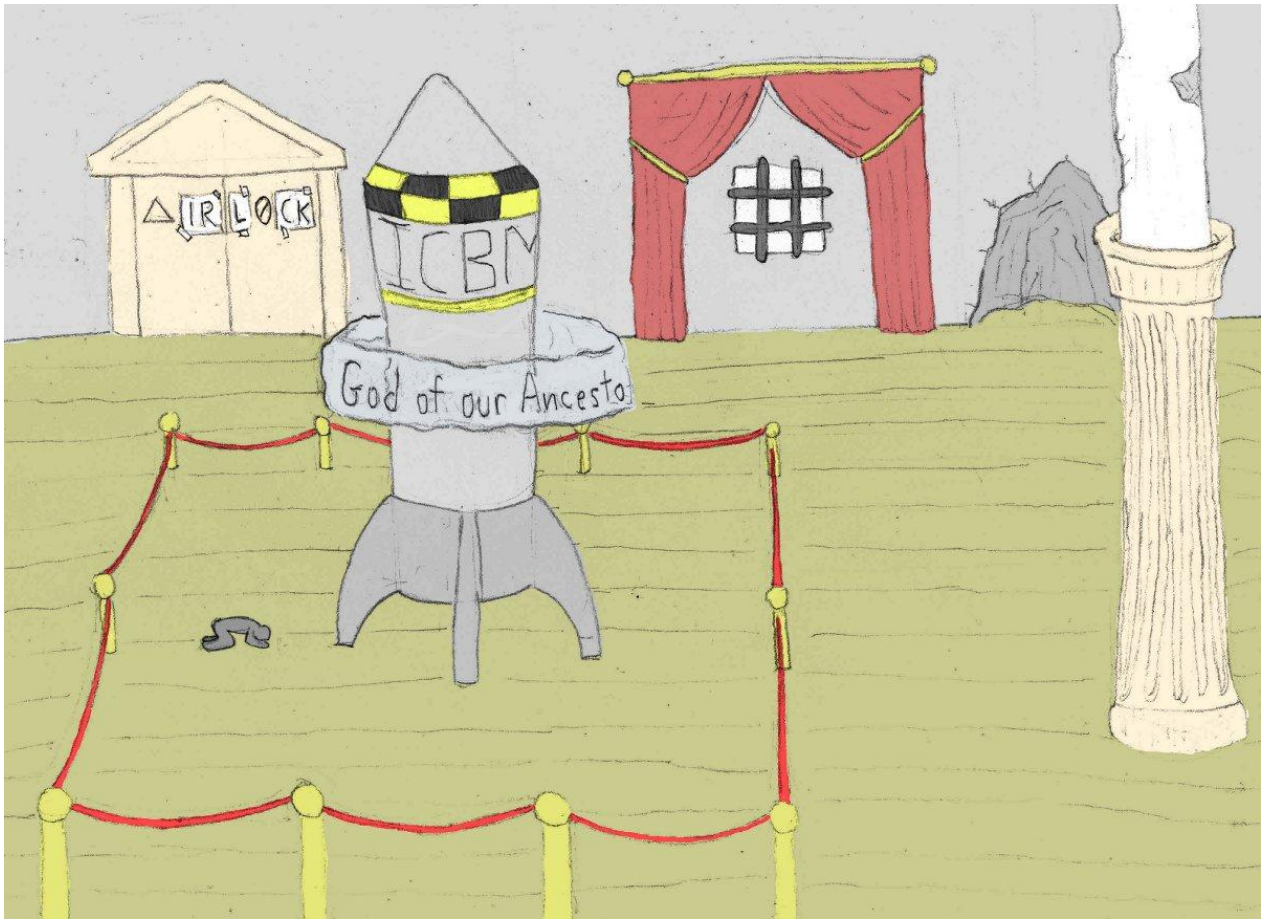
http://www.jewell.edu/william_jewell/gen/william_and_jewell_generated_bin/images/basic_module/track.jpg

http://www.paranormalknowledge.com/wp-content/uploads/2009/07/high-school-hallway-300x224.jpg

The Society for Historical Beatdowns Wing

This wing is located behind the Academy wing inside of a large, two-story tall room. It constructed as a museum, and is filled with relics of our military history; however, their purposes of are grossly

misinterpreted. The general look of this wing is one of cheaply purchased luxury obscuring serious faults in the building. Columns hide crumbling supports. Velvet curtains hide destroyed sections of wall. The only windows in the wing are covered in heavy, protective steel bars. However, despite all of this, the professors are immensely proud of their creation.

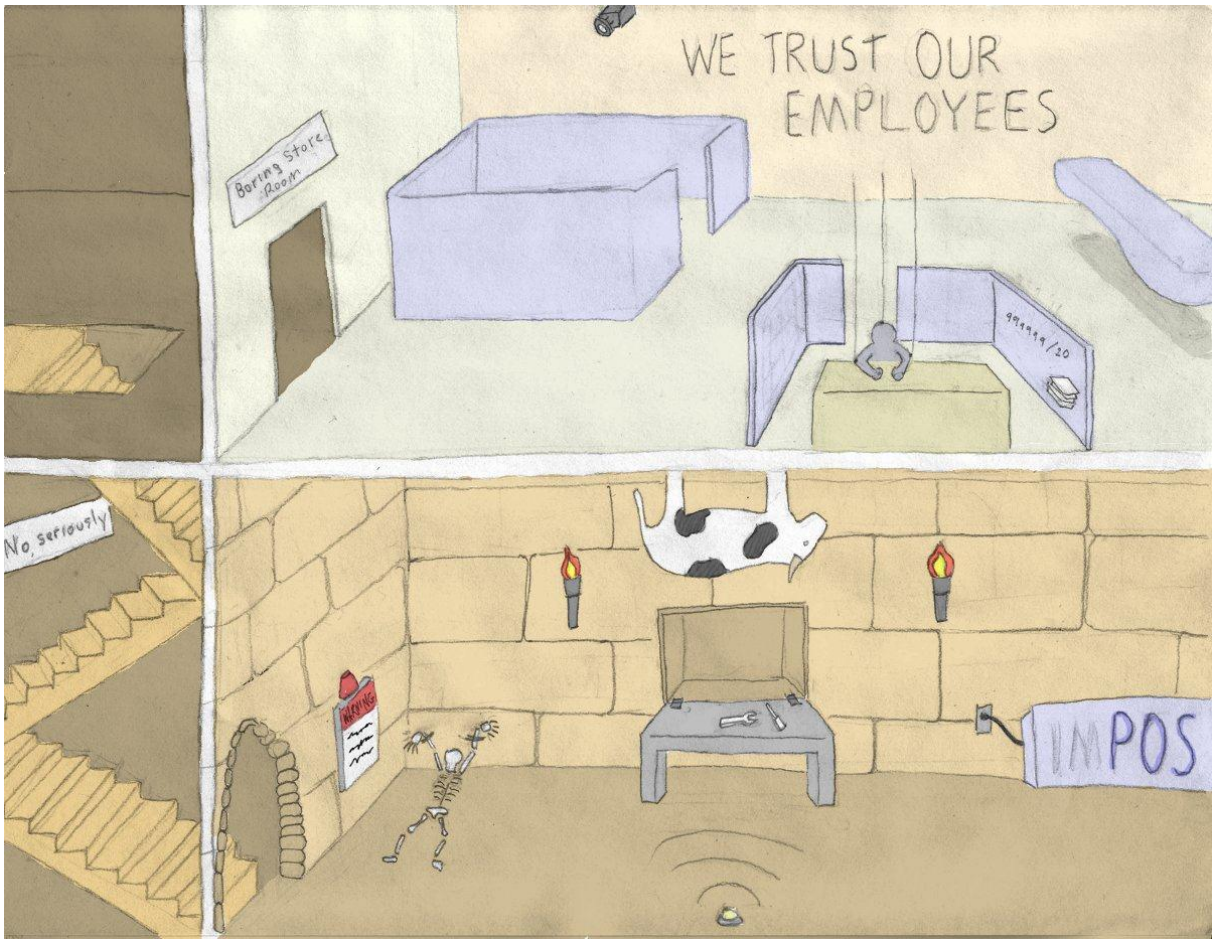


References

http://www.lensimpressions.net/index.php?showimage=515

The Impossible Possibilities Wing

The Impossible Possibilities Wing has a split personality. Upstairs it is the only futuristic looking location in the entire academy. Materials like glass and stainless steel dominate. Object float, hang, and sparkle in the sun. Even the cubicles are built from transparent displays. However, this luxury comes at a price. The entire floor is under heavy, automated observation by the parent corporation. Big brother is present in all of the hidden and not so hidden niches. Luckily, the lab staff and students have a place where they can work unobstructed. The basement of this wing taps into the dungeons of the fort. In this spooky environment, modern technology mixes with ancient stockades. The combined look is an accidental mad scientist's lab set in a dimly lit underground cavity.



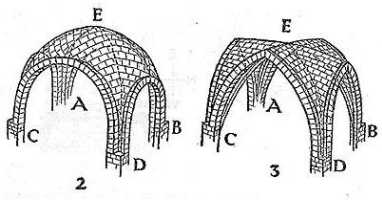
References

Ground Level Cubicles

http://www.peterallenco.com/DSCN0095.JPG

Basement Lab

http://dolinsky.fa.indiana.edu/caveart/spr09/jcterzin/jct_dungeon1.jpg



Scenes

Test Render: The Sorting Room

For details on the layout, see the Models and Textures section.

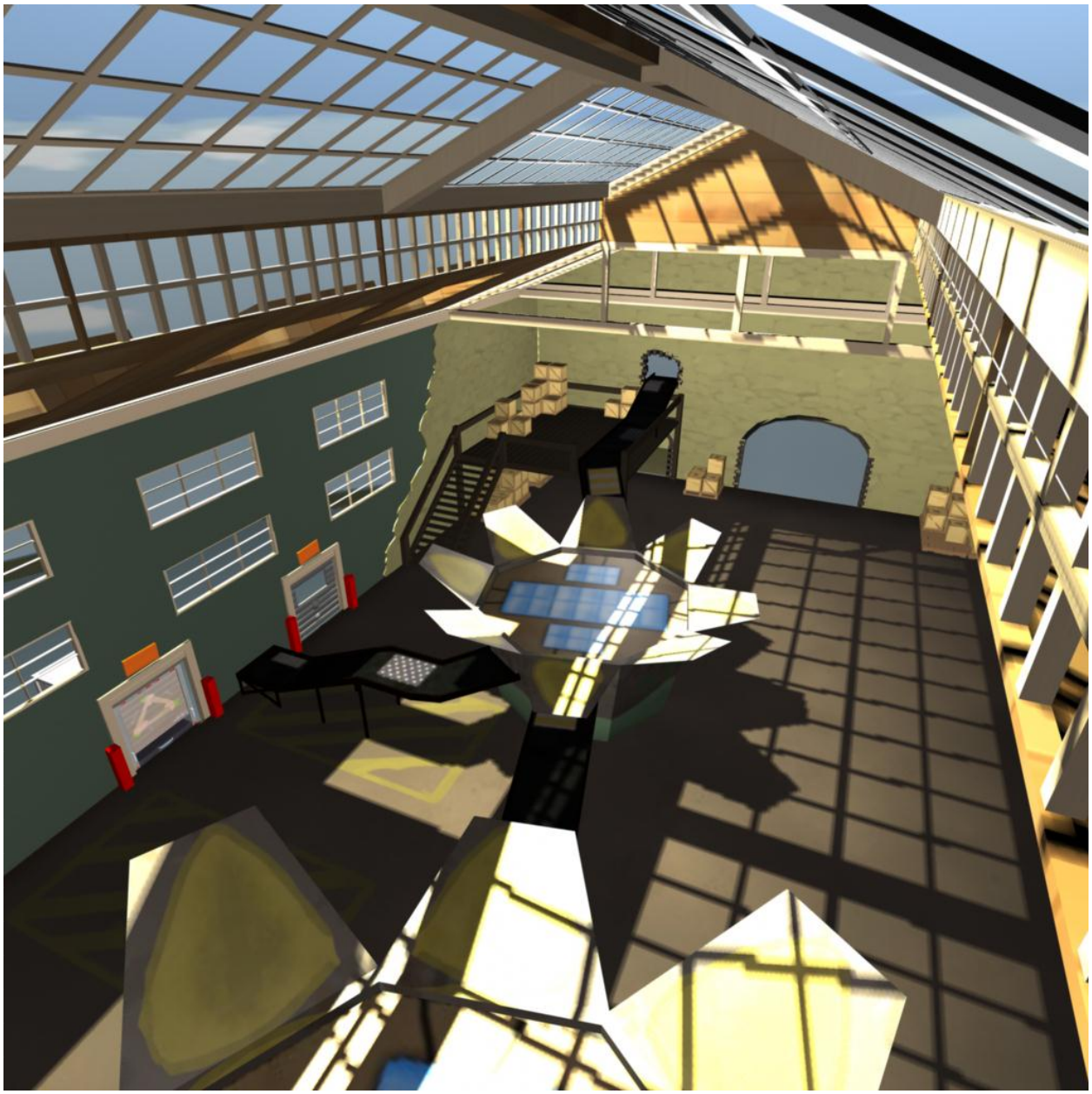


Figure 34: Wide shot from the ceiling.

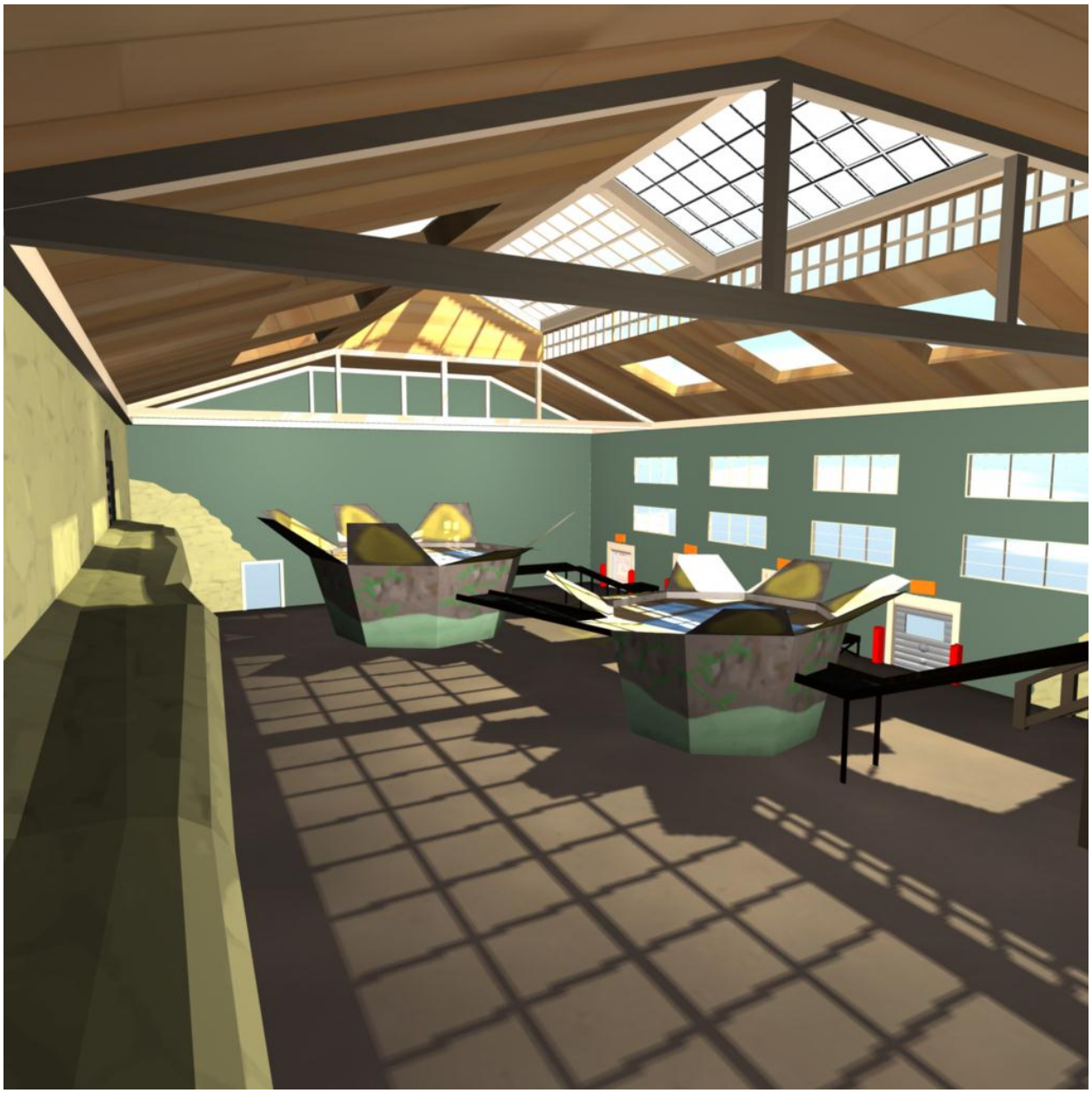


Figure 35: From the ledge.

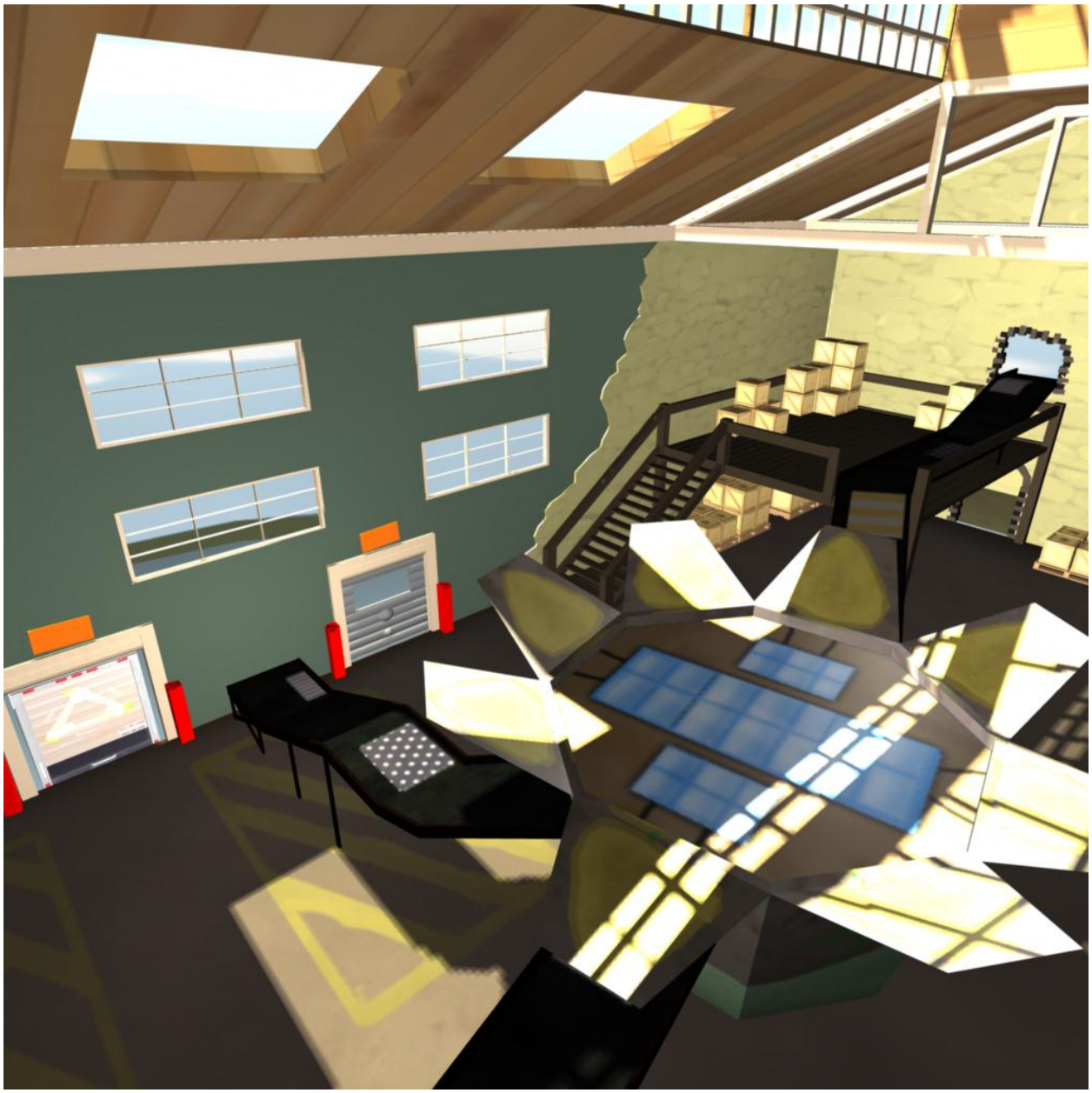


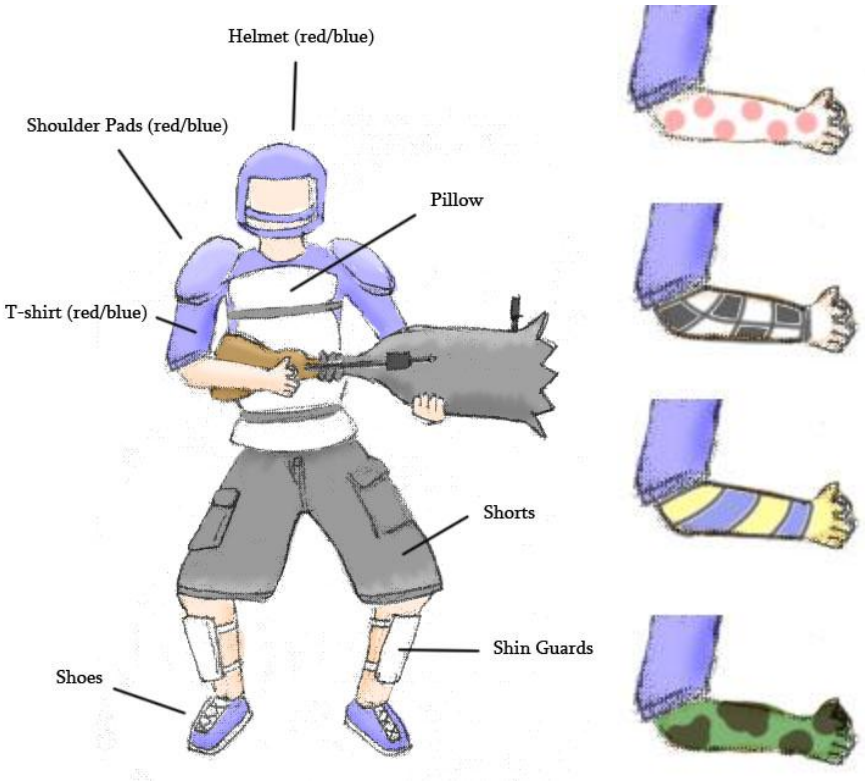
Figure 36: The sorting machine and conveyors.

Models and Textures

Each asset is rated according to three criteria:

- Gameplay Importance [5 most – 1 least] – How much losing this asset would affect the gameplay (level flow, not having a weapon to fire, etc.).
- Art Importance [5 most – 1 least] – How much losing this asset would affect the ability of the game to convey story and style.
- Simplifiable? - Can the geometry and textures be simplified or is it critical that this asset appear exactly as stated.

Player Avatar



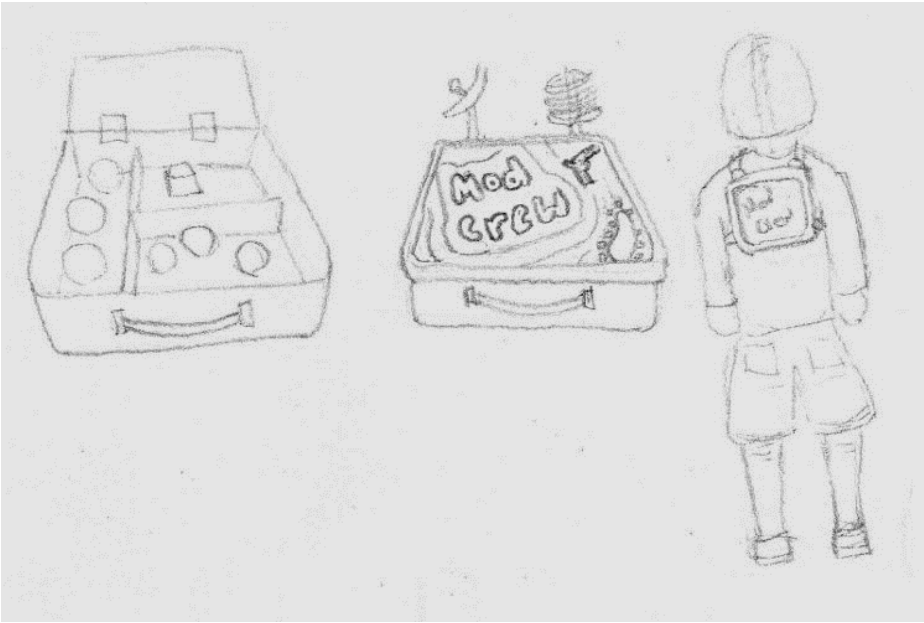
- Meshes
  - A medium-build humanoid with shoes and shin guards
  - A t-shirt attached to torso
  - A pair of shorts attached to lower body
  - A set of shoulder pads
  - A football style helmet
  - A pillow attached to the torso
- Textures
  - Body parts
    - Left Arm
    - Right Arm

        - Left Leg
        - Right Leg
        - Head
        - Hand
    - Team Colored Items (red and blue coloring)
        - Helmet
        - Shoulder Pads
        - T-shirt
        - Shoes
    - General Items (one coloring)
        - Grey Shorts
        - White pillow
        - White Shin guards

Weapons

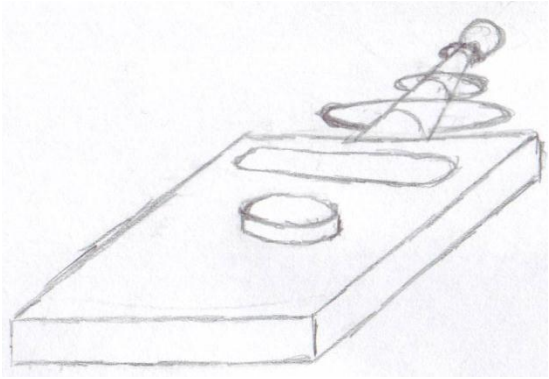
Modifier Launcher

- *Description:* A metal lunch box like container that holds the modifiers. Decorated with the "Mod Crew" logo (70s theme) and various technical components (radar dish, etc.).
- *Quality:* High.
- *Use*: Fires the modifiers.
- *Importance:* Gameplay (5). Art (3). Can be simplified.



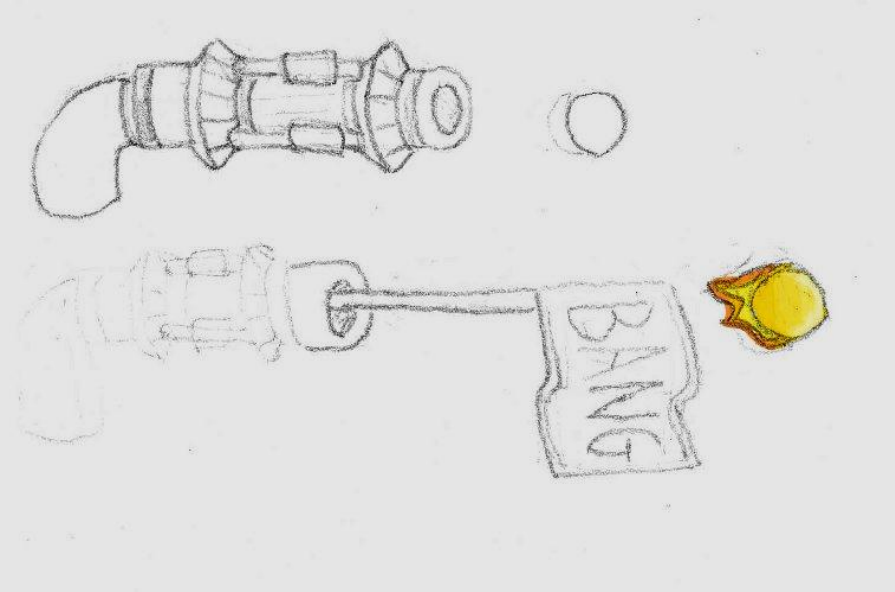
Nullifier Weapon

- *Description:* A remote control-like device made from a pocket calculator and an antenna. The button is large and red.
- *Quality:* High.
- *Use*: Held in one hand by the player.
- *Importance:* Gameplay (5). Art (3). Can be simplified to a remote with only a single button.



Pistol "The Small Bang"

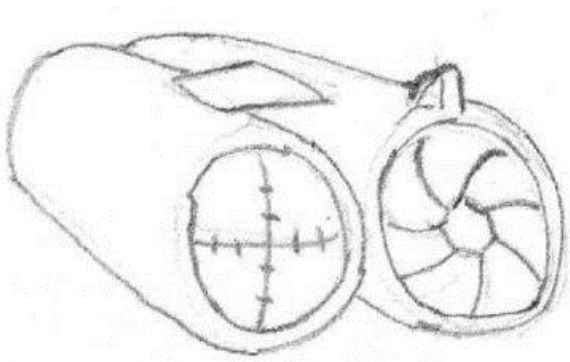
- *Description:* A modified pop gun. Augmented with small pistons to work the pumping motion. Constructed from a fine grained wood material.
- *Quality:* High.
- *Use*: Shoots normal and flaming ping pong balls.
- *Importance*: Gameplay (5). Art (4). Can be simplified to a basic pop gun.



Sniper Rifle "Binocular Sniper Rifle"

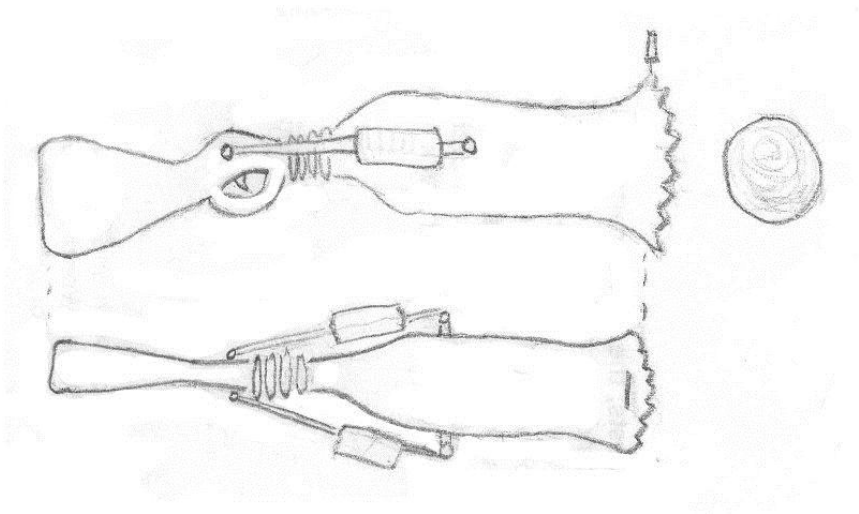
- *Description:* A compact sniper rifle mounted inside a pair of binoculars. The left ocular contains the barrel while the right ocular contains the scope. There are two triggers, one on each ocular where the hands hold.

- *Quality:* High.
- *Use*: Held with two hands. Each on the triggers of the binocular scopes.
- *Importance*: Gameplay (5). Art (3). Can be simplified a small amount, but the scope and barrel much remain.



Explosive Projectile Weapon "Cannon Gun"

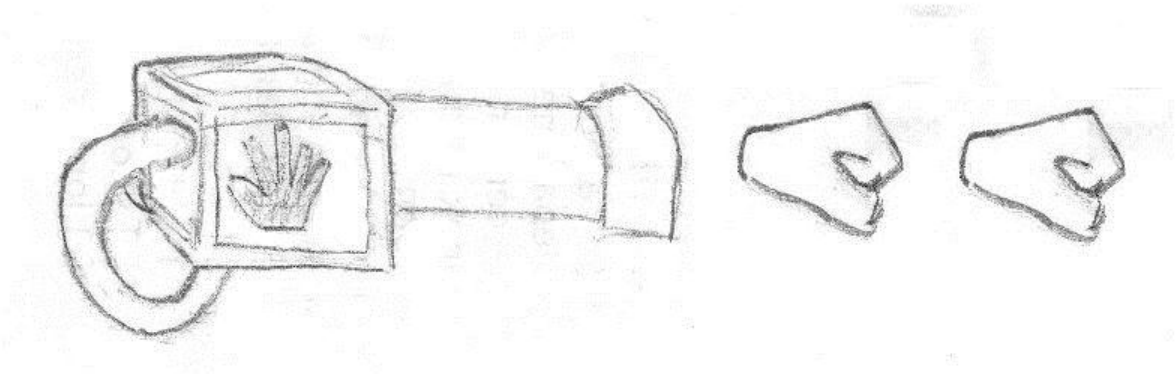
- *Description:* A blunderbuss widened at the end to fire explosive cannon balls. Augmented with shock absorbers, one on each side and a joint spring in the middle. Features a large crosshair sight on the end of the barrel.
- *Quality:* High.
- *Use*: Held with two hands. One on the trigger, one on the stock.
- *Importance*: Gameplay (5). Art (3). Can be simplified as long as it maintains its shape and joint spring.



Assault Rifle "Fists of Fury"

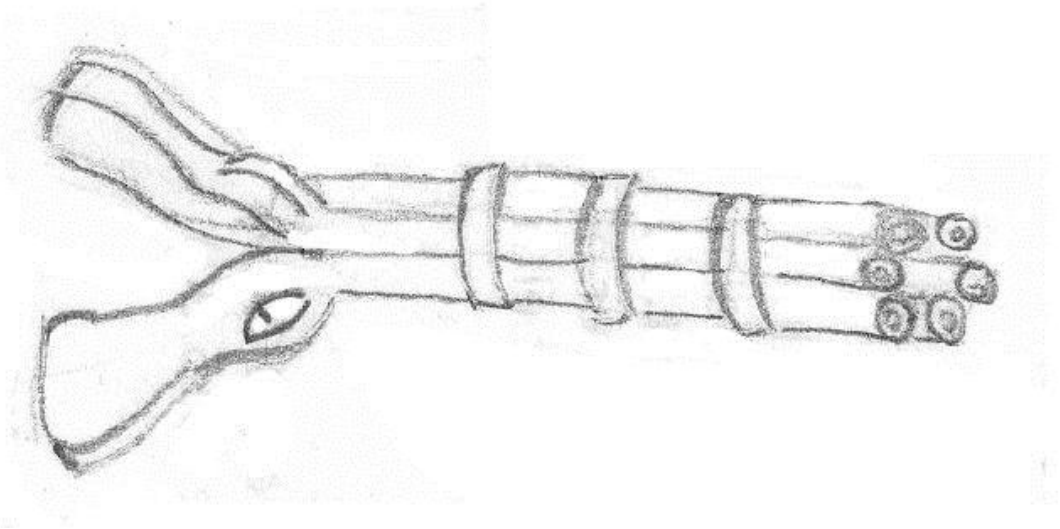
- *Description:* A gun that fires unusable hands in the form of fists. It feeds from a parts box and projects them from a steel tube.
- *Quality:* High.
- *Use*: Held with two hands. One on the stock, one on the trigger.

- *Importance*: Gameplay (5). Art (4). Cannot be simplified.



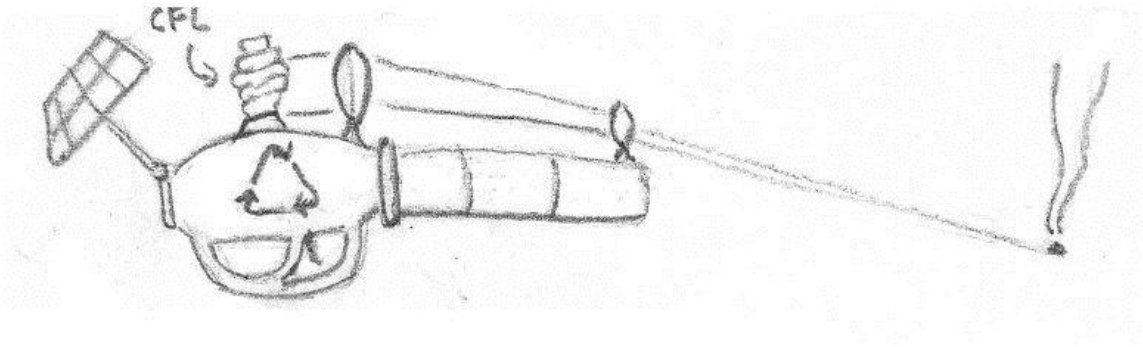
Shotgun "Multi-Shot"

- *Description:* Many single shot guns taped together. They reload at different frequencies.
- *Quality:* High.
- *Use*: Held with two hands. One on the stock, one on the trigger.
- *Importance*: Gameplay (5). Art (2). Can be simplified or outright changed.



Damage over Time Weapon "The Composter"

- *Description:* A solar powered heat ray gun, powered by an energy efficient CFL. Over-driven version of a tool used for composting and moving soil. A solar panel is mounted on its back; while an array of focusing lenses sit on the top.
- *Quality:* High.
- *Use*: Held with two hands. One on the stock, one on the trigger.
- *Importance*: Gameplay (5). Art (4). Can only be simplified very little without losing its meaning.



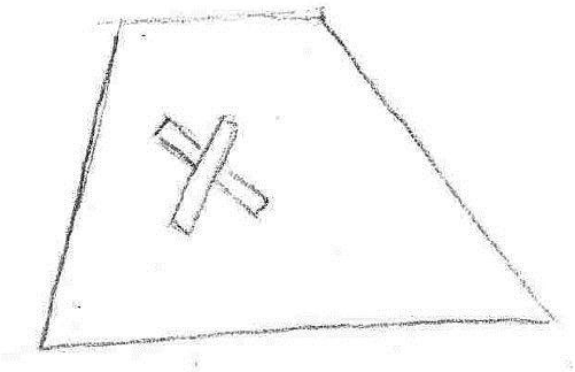
Grenade "Pop Can"

- *Description:* A soda can filled with explosives. Triggers shortly after the tab is popped. Explodes in a fireball.
- *Quality:* High.
- *Use*: Held with one hand.
- *Importance*: Gameplay (5). Art (2). Cannot be simplified.

Modifiers

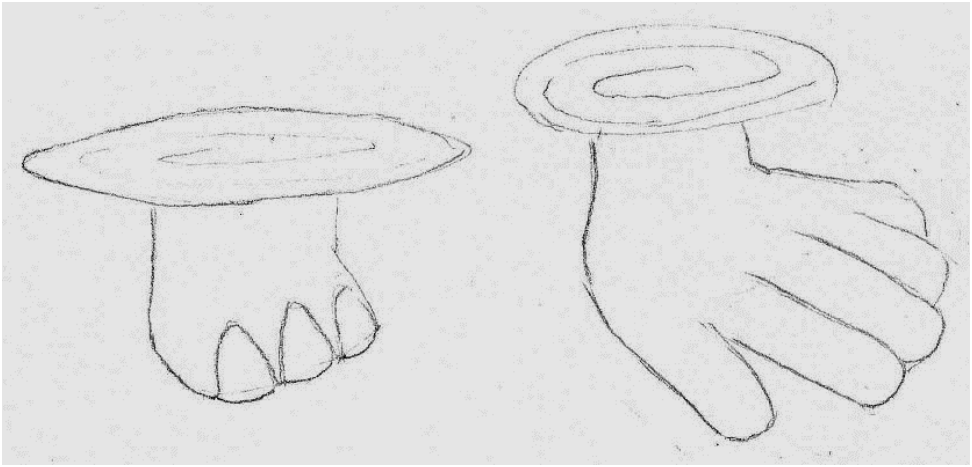
Modifier Marker

- *Description:* An X shaped decal marking where a modifier is. Colored to almost but not quite blend in with the environment. Appears to be made from tape.
- *Quality:* High.
- *Use*: Placed to mark where a modifier is deployed.
- *Importance*: Gameplay (5). Art (2). Cannot be simplified.



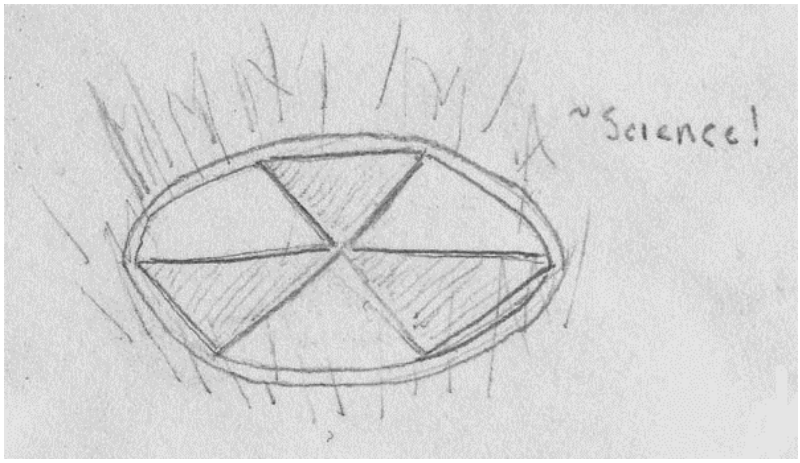
Shrink Effect

- *Description:* A portal appears 6 feet along the normal of the surface that the modifier is on. A large hand or dinosaur foot stomps downward crushing the player for the duration. The portal features a swirl distortion and is textured like a star field.
- *Quality:* High.
- *Use*: Triggered when a Shrink modifier goes off.
- *Importance:* Gameplay (5). Art (4). Can be simplified as long as it intuitively represents crushing or shrinking.



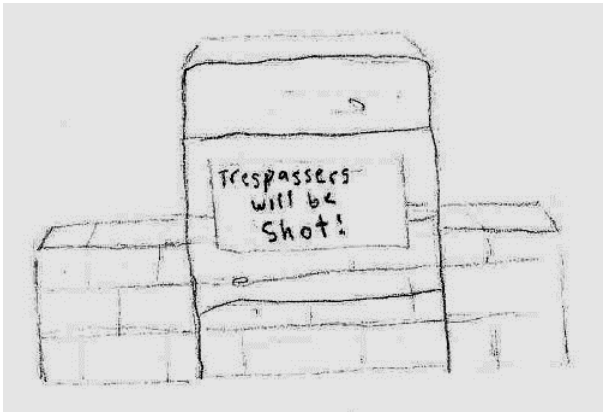
Growth Effect

- *Description:* A large radiation symbol glowing with unnatural orange light. Indicative of mutating radiation.
- *Quality:* High.
- *Use*: Appears when the Grow modification is triggered. Causes the player model to get bigger as described in the Grow animation.
- *Importance:* Gameplay (3). Art (4). Can be simplified as long as it communicates the idea of growth.



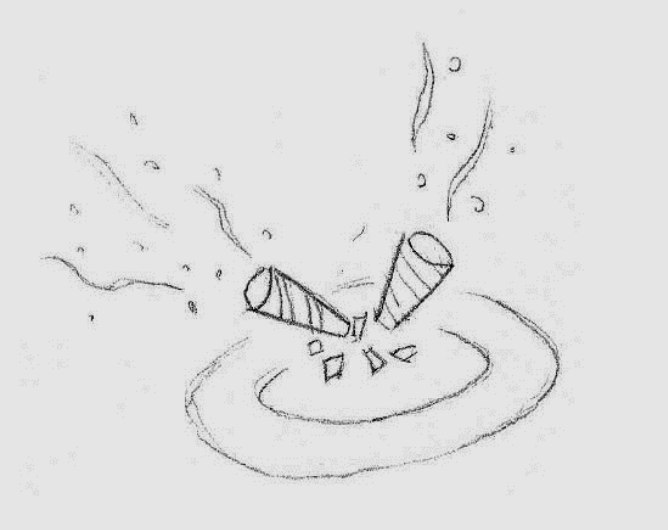
Wall Effect

- *Description:* A wall make of transparent wood planks. It is as tall as a player and twice as wide. Has depth equal to twice a player's depth. An opaque sign on the front says "Trespassers will be Shot." It pops up from the modifier's surface along the surface normal.
- *Quality:* High.
- *Use*: Triggered when a wall modifier is activated.
- *Importance*: Gameplay (5). Art (3). Can be simplified as long as it blocks the player's path but not their view.



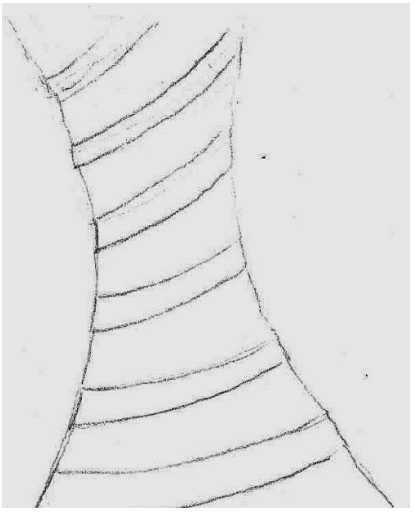
Knockback Effect

- *Description:* An explosion of confetti and streamers.
- *Quality:* High.
- *Use*: Triggered when the knockback modifier is activated.
- *Importance*: Gameplay (3). Art (3). Can be simplified as long as it communicates the idea of an explosion.



Gravity Inversion Effect

- *Description:* A wormhole-like swirl extends from the modifier location upwards along the current gravity vector. Animates to show the swirl like a small, thin disk moving up a barbershop pole.
- *Quality:* High.
- *Use*: Triggered when a gravity inversion modifier is activated.
- *Importance*: Gameplay (4). Art (3). Can be simplified as long as it points the direction of the player's movement.

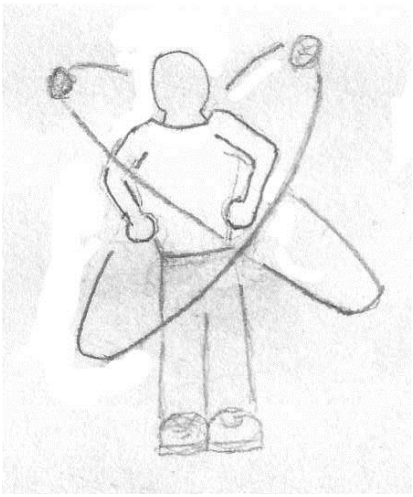


Frictionless Effect

- *Description:* Replace the player's lower body clothing with a pair of footie pajamas.
- *Quality:* High.
- *Use*: Swapped into the model texture when the player is under the effect of the Frictionless modifier. Does not interrupt the player's running leg animations.
- *Importance:* Gameplay (5). Art (2). Cannot be simplified.

Increase Gravity Effect

- *Description:* Two planetoid-like objects circle the player in perpendicular orbits.
- *Quality:* High.
- *Use*: Appear when the player is under the effect of the Increase Gravity effect.
- *Importance*: Gameplay (5). Art (3). Can be simplified as long as it communicates increased weight.



Illusion Effect

- *Description:* Spawns several copies of the player's model to distract enemies. Appears just like a player avatar.
- *Quality:* High.
- *Use*: Appear when the Decoy modifier is activated.
- *Importance*: Gameplay (5). Art (3). Cannot be simplified.

Consult the Player Avatar section for concept art.

Map: The Academy

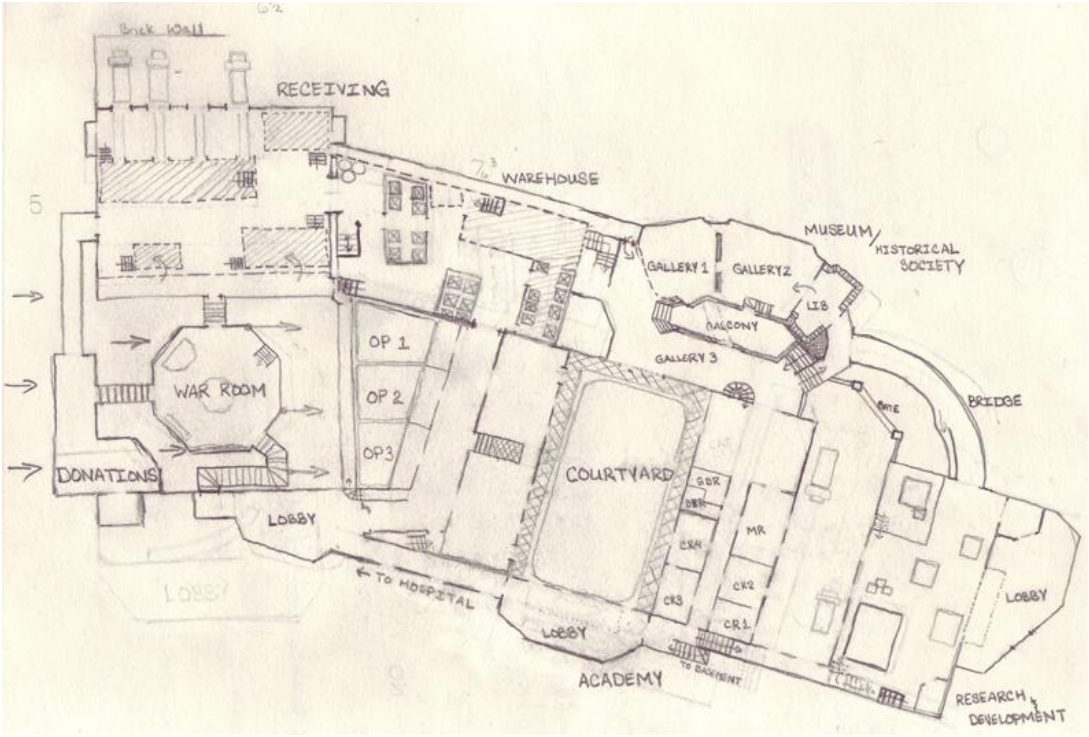


Figure 37: Blueprint for Academy Level

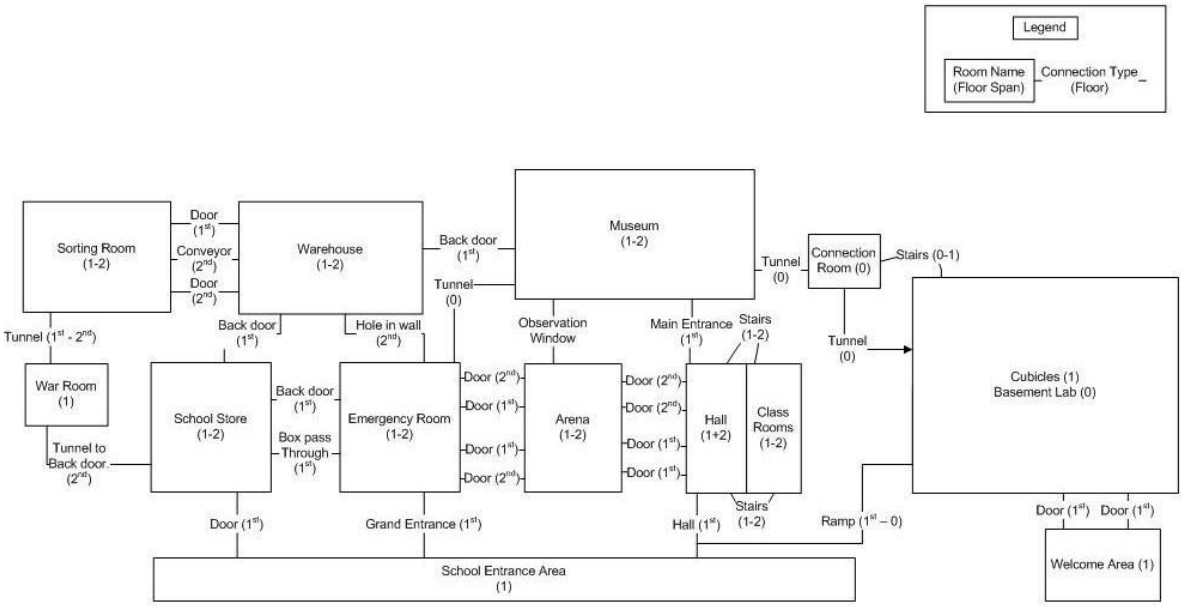


Figure 38: Connection Flowchart for each room in the Academy Level

The Appendage Conservation Front Wing

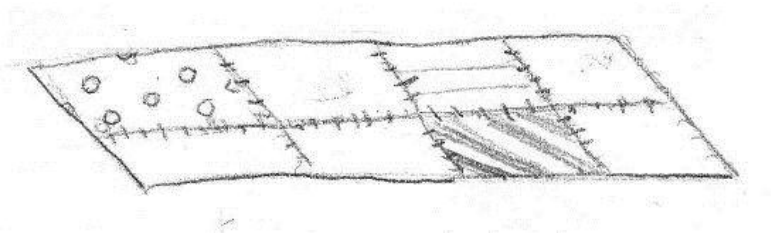
Sorting Room

Environment

This room is open, airy, and has a cheerful tenor. It is lit through its myriad windows and is painted in natural colors. The floor is dirty, but not damaged or neglected. The walls are half natural material, half old stone from the fort. Vines and moss cover parts of it.

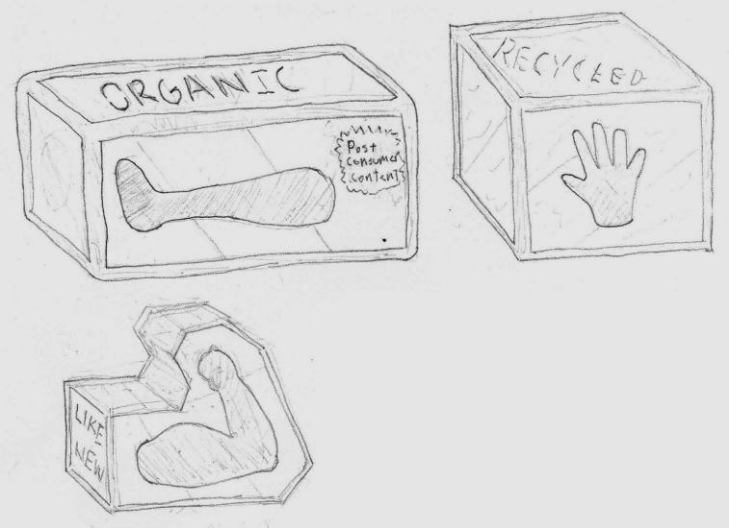
Conveyor Belt

- *Description:* Several belts at varying levels of inclination. Fairly worn down. Wide enough to support two players side by side. The belt has worn thin in places and been fixed up with cloth patches.
- *Quality:* Medium, 250/piece.
- *Use*: Proceed into and out of the sorting machine. Used as walkways.
- *Importance:* Gameplay (4). Art (3). Can be simplified in texture.



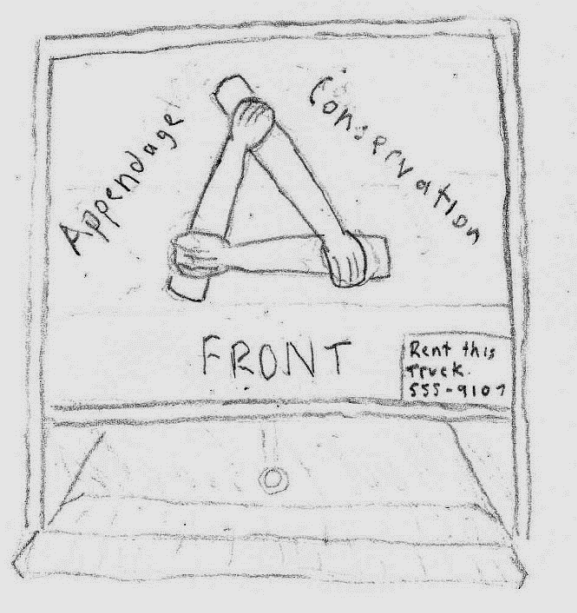
Wooden Body Part Boxes

- *Description:* Wooden boxes with depictions of appendages on them. At least one depiction for upper body and one for lower body.
- *Quality:* Low, 12-50.
- *Use*: Placed on conveyor belts and in stockpiles.
- *Importance*: Gameplay (3). Art (5). Cannot be simplified.



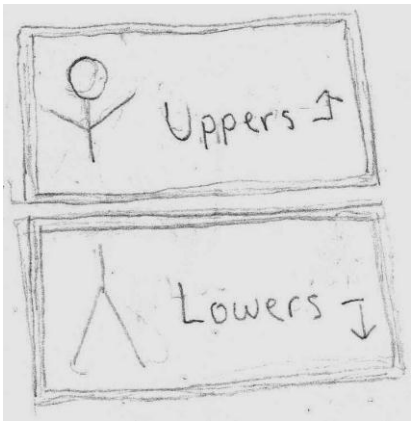
Delivery Truck Rear

- *Description:* The back of a delivery truck, partly open but not accessible. Decorated with the ACF logo.
- *Quality:* Low-Medium, 35-50.
- *Use*: Placed along the wall with only the rear two feet showing.
- *Importance*: Gameplay (1). Art (3). Can be simplified in complexity or texture.



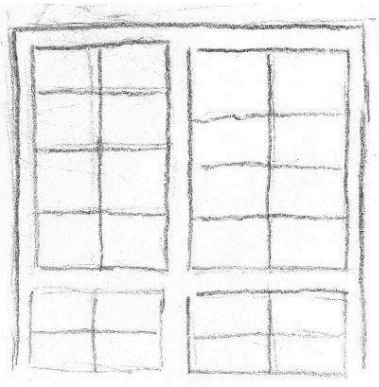
Categorization Signs

- *Description:* Two signs identifying the two categories that parts are sorted into: "Uppers" and "Lowers". Hand-painted.
- *Quality:* Low, 2-26/piece.
- *Use*: Placed above the conveyor exits from the *Sorting Room* into the *Warehouse*.
- *Importance*: Gameplay (1). Art (4). Can be simplified as long as body depictions are preserved.



Windows

- *Description:* Large, greenhouse like window with multiple panes.
- *Quality:* Low, 2-18.
- *Use*: Allows light to enter through the backside and top of the sorting room.
- *Importance*: Gameplay (1). Art (3). Cannot be simplified.

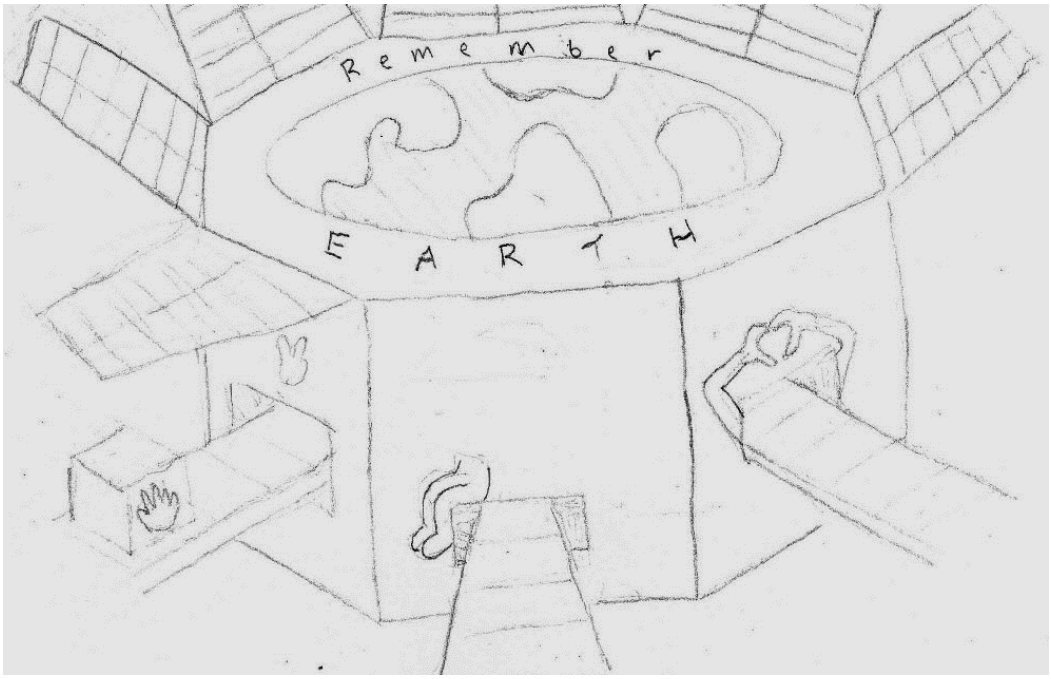


Flower Decorations "Diversi-pot"

- *Description:* Combination of several flower types on a single stem. Held in a small red pot. Should be scalable to two different heights (2' and 4').
- *Quality:* Medium, 800-1000.
- *Use*: Used to decorate and freshen the borders of the room.
- *Importance*: Gameplay (1). Art (3). Can be simplified as long as the stem holds at least two distinct flowers/fruits.

Sorting Machine

- *Description:* Large octagonal machine for sorting parts. Flat on the top with solar panels arranged around the outside like flowers petals. Various openings for conveyors to enter or exit. Decorated with environmental graffiti.
- *Quality:* Medium, 500-800.
- *Use*: Used as a platform on which to fight.
- *Importance*: Gameplay (5). Art (4). Textures can be simplified.



Warehouse

Environment

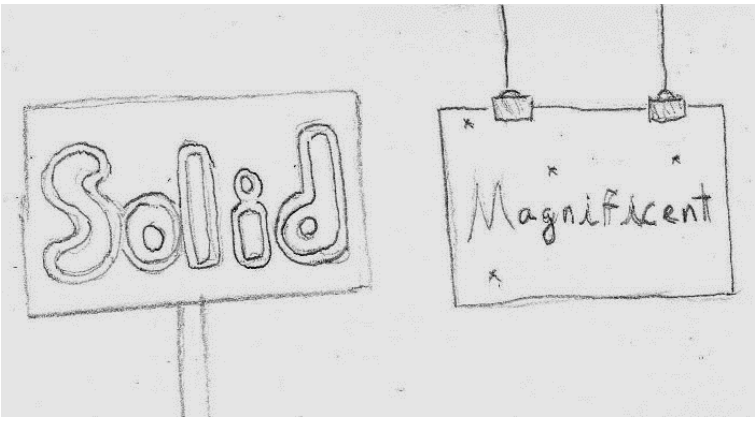
Large, cavernous, stone room that used to function as the armory. Worn down feeling. Some of the walls have moss on them.

Wooden Body Part Boxes

- *Defined In:* "Wooden Body Part Boxes" in *Appendage Conservation Front - Sorting Room*.

Categorization Signs

- *Description:* Signs labeling the various stock piles in the warehouse. Cardboard, hand painted. Size Labels: "Scrappy," "Solid." Types: "Hairy", "Goofy", "Magnificent," "Whoa"
- *Quality:* Low, 20-40/piece.
- *Use*: Placed above the stock piles, either hung or on a pole projecting from a pile.
- *Importance:* Gameplay (1). Art (4). Can be simplified a bit in texture.



Armory Label

- *Description:* An official, solid looking military sign saying "ARMORY" with quotes around "ARM" (the joke being that it houses arms...). Yellow and black warning stripes on the sides.
- *Quality:* Low, 2-10.
- *Use*: Large scale and place on the longest wall of the room at a height to allow viewing from all parts of the room.
- *Importance*: Gameplay (1). Art (3). Cannot be simplified.

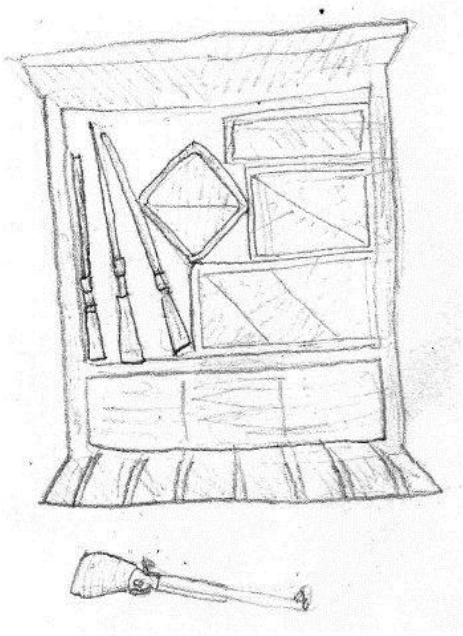
Warning Sign

- *Description:* A conspicuous sign warning of the dangers of discharging live ammunition in the Armory. Official, standardized look. Full of bullet holes and dents.
- *Quality:* Low, 2-10.
- *Use*: Placed around doors and stockpiles.

- *Importance*: Gameplay (1). Art (4). Cannot be simplified.

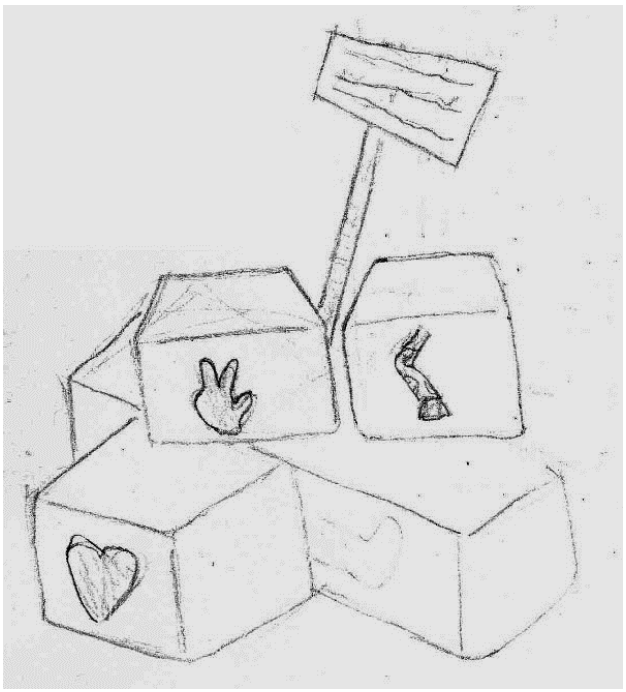
Gun Racks

- *Description:* Basic, functional racks that can hold a number of weapons upright. Overflowing with both boxes and weapons.
- *Quality:* Low, 60 w/o objects.
- *Use*: Allow players to run along top. Placed in series across the room.
- *Importance:* Gameplay (5). Art (3). Can be simplified to unadorned warehouse pallet racks.



Part Piles

- *Description:* Messy collections of part boxes labeled with signs.
- *Quality:* Medium, 100-200.
- *Use*: Placed on the floor around the room.
- *Importance*: Gameplay (4). Art (4). Cannot be simplified.



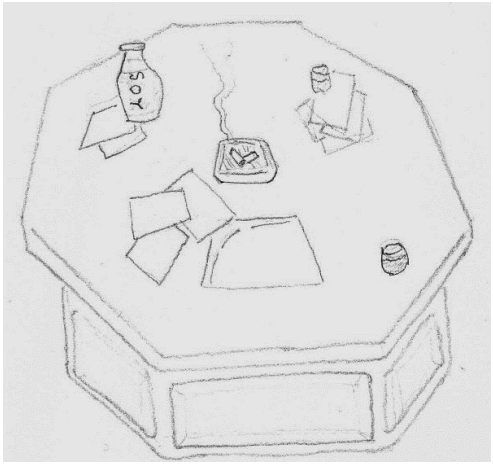
War Room

Environment

Octagonal shape. Wood paneled walls. Tile floor. Has a dim, smoky feel.

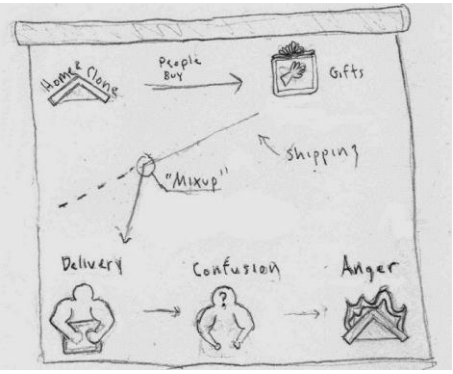
War Table

- *Description:* An octagonal table covered with paperwork, a cigarette tray, a bottle of soy-milk, and scotch glasses.
- *Quality:* Medium, 200-300.
- *Use*: Placed in the middle of the war room.
- *Importance*: Gameplay (3). Art (4). The items on top of the table can be simplified or removed.



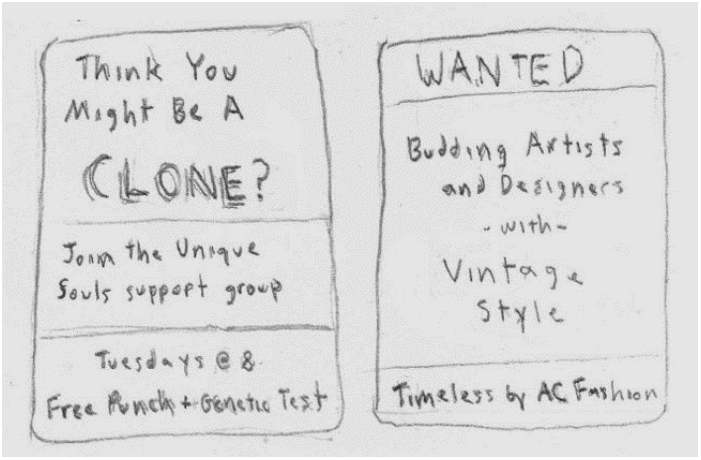
Battlefield Map

- *Description:* A gridded map depicting an upcoming operation. The map shows the target ("Home & Clone"), their shipping network, the plan to swap packages to mismatch gift deliveries, and the result (a large muscular guy now with tiny forearms).
- *Quality:* Medium-High, 150.
- *Use*: Hung prominently on one wall.
- *Importance*: Gameplay (1). Art (4). The map details can be simplified to a single target.



Wanted Posters

- *Description:* Various posters advertising ACF services: support group for clones, and a recruitment poster for their vintage fashion line.
- *Quality:* Low, 2-10.
- *Use*: Placed around the room.
- *Importance*: Gameplay (1). Art (3). Cannot be simplified.



School Store

Environment

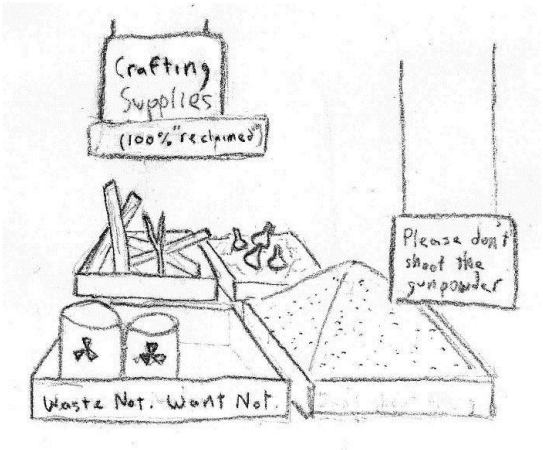
Carpeted floors. Walls painted green on the bottom half. Parts of the walls are covered with banners for the school. It is split into two vertical levels, with the left side of the store raised higher than the right. The left side has school supplies while the right has parts.

Bargain Bin

- *Description:* An 8' x 8', low-sitting bin with parts boxes piled in it. This bin connects through a chute horizontally to the emergency room.
- *Quality:* Medium, 200-300.
- *Use*: Placed near the center of the store.
- *Importance*: Gameplay (4). Art (4). Cannot be simplified.

Reclaimed Crafting Supplies Bins

- *Description:* Bins containing recycled materials including: gun powder, radioactive waste, wood, and assorted chemicals.
- *Quality:* Low, 200-300.
- *Use*: Placed near the entrance to the store (interspersed with school supplies).
- *Importance:* Gameplay (2). Art (3). Can remove some of the supplies types.



Register

- *Description:* Registers for selling the various supplies at the store. Marked with signs advertising for trade-ins.
- *Quality:* Low, 50.
- *Use*: Placed near the entrance to the store.
- *Importance*: Gameplay (1). Art (3). Details can be simplified.

Categorization Signs

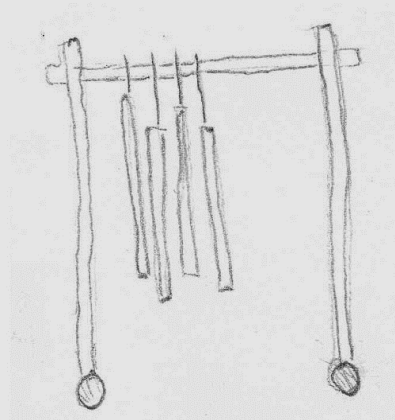
- *Description:* Signs labeling the different part types being sold. Roles include: Strong arms for holding weapons, Heat resistant arms for crafting, glowing appendages for drawing attention, and Camouflaged parts for blending in.
- *Quality:* Low, 2-10.
- *Use*: Placed above the mannequin displays below.
- *Importance:* Gameplay (1). Art (3). Cannot be simplified.

Mannequin Displays

- *Description:* Mannequins on platforms showing off the looks specified in "Categorization Signs."
- *Quality:* Medium.
- *Use*: Spread about the room.
- *Importance:* Gameplay (3). Art (4). Can reduce the number of displays or the scene complexity.

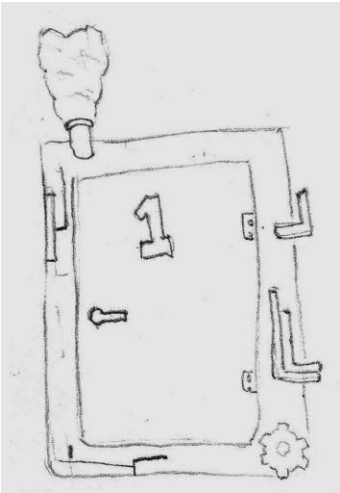
Clothing Racks

- *Description:* Generic clothing racks holding shirts of various sizes. New clothes to fit the shape and style of your new parts.
- *Quality:* Medium, 250-300 w/o items.
- *Use*: Used to segment the room. Placed around the exterior.
- *Importance:* Gameplay (4). Art (3). Cannot be simplified.



Dressing Room Door Facings

- *Description:* Doors that are decorated and numbered like dressing room doors, with additional machinery around the periphery to indicate the technological process of switching parts.
- *Quality:* Medium, 400-500.
- *Use*: Placed in a row against one wall.
- *Importance*: Gameplay (1). Art (4). The ornamentation of the facing can reduced.



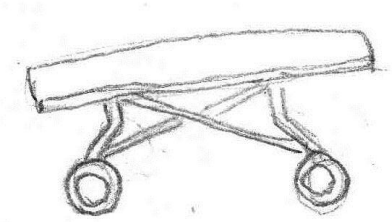
Emergency Room

Environment

Hospital white floors. Well lit. Sectioned off by curtains. Light green highlights on the walls.

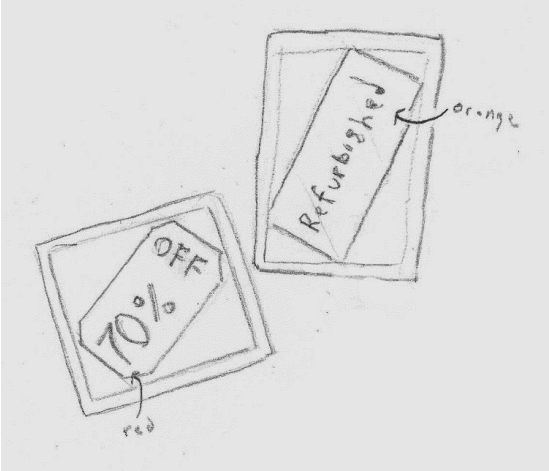
Gurney

- *Description:* A standard, while hospital gurney.
- *Quality:* Medium, 400-500.
- *Use*: Placed by the entrance doorway.
- *Importance*: Gameplay (2). Art (4). Cannot be simplified.



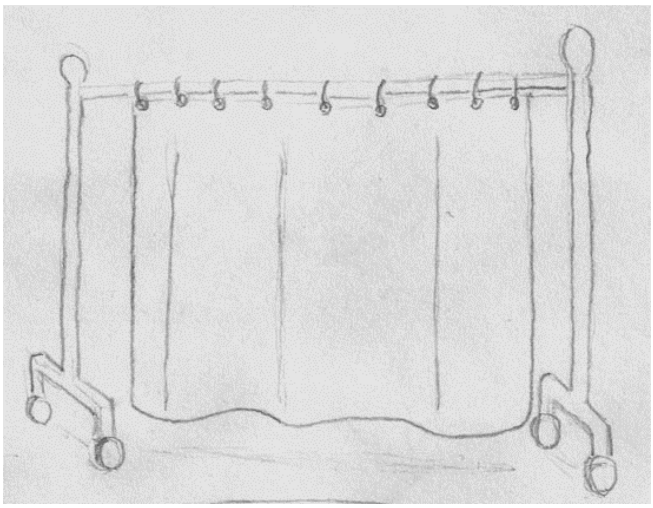
Marked Down Parts Boxes

- *Description:* A parts box covered with orange stickers that announce its low price or damage goods.
- *Quality:* Medium, 400-500.
- *Use*: Placed in the hopper holding boxes that came from the school store bargain bin.
- *Uses Content From:* Parts Boxes - *Appendage Conservation Front Wing - Sorting Room*
- *Importance:* Gameplay (3). Art (4). Cannot be simplified.



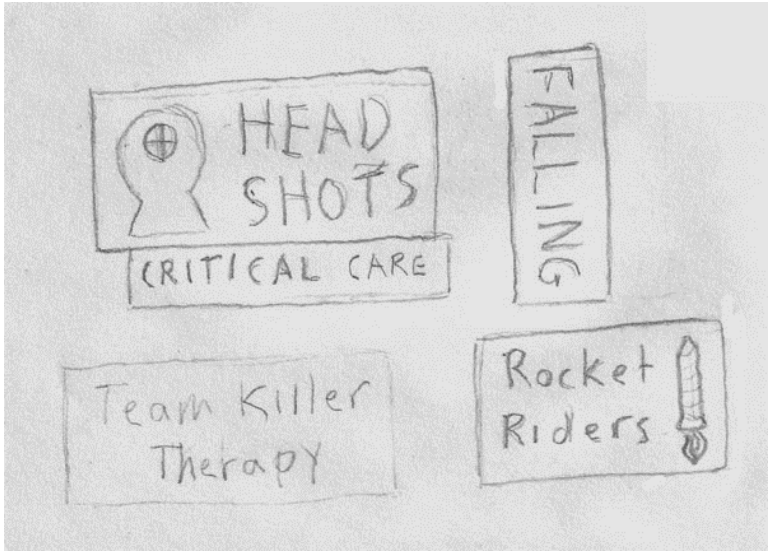
Curtains

- *Description:* Light blue curtains hung from rolling metal racks.
- *Quality:* Medium
- *Use*: Breaks the emergency room into different areas of specialty.
- *Importance:* Gameplay (4). Art (4). Cannot be simplified.



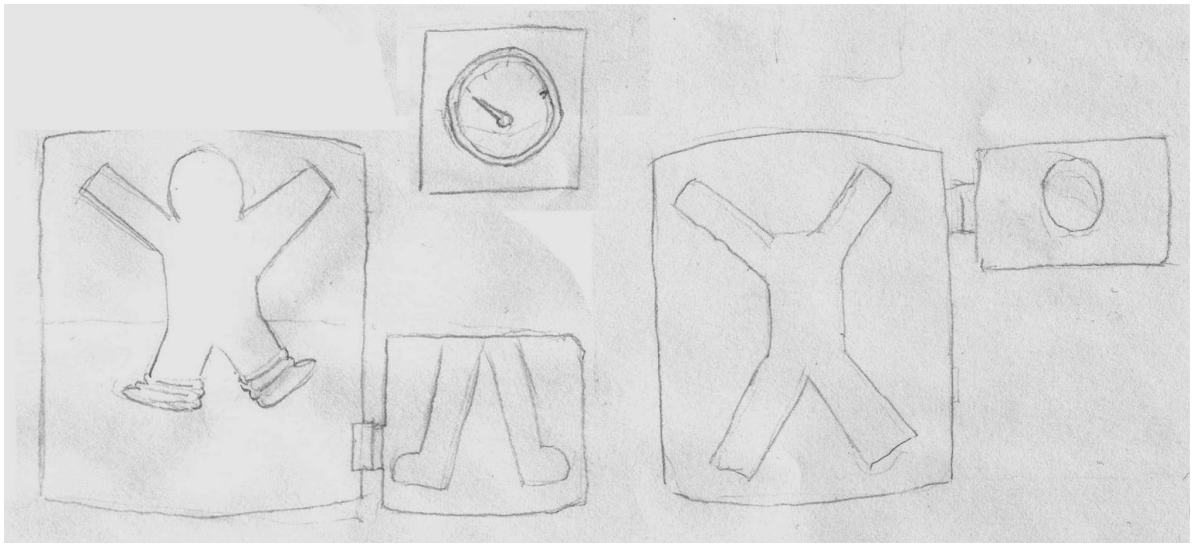
Ward Signs

- *Description:* Signs marking the different specialty areas in the emergency room.
- *Quality:* Medium
- *Use*: Placed on the walls and curtains of the different areas.
- *Importance*: Gameplay (1). Art (4). Number of wards can be reduced.



Respawners

- *Description:* Specialized machinery for quickly repairing FPS victims ("Out in 5 seconds or less"). Shaped to represent the different injuries. A respawn timer is placed on the outside of the door.
- *Quality:* High.
- *Use*: Placed around the room within the curtains. They lay on the floor in the same orientation as an operating table.
- *Importance*: Gameplay (2). Art (4). Number of types can be reduced.



Academy Wing

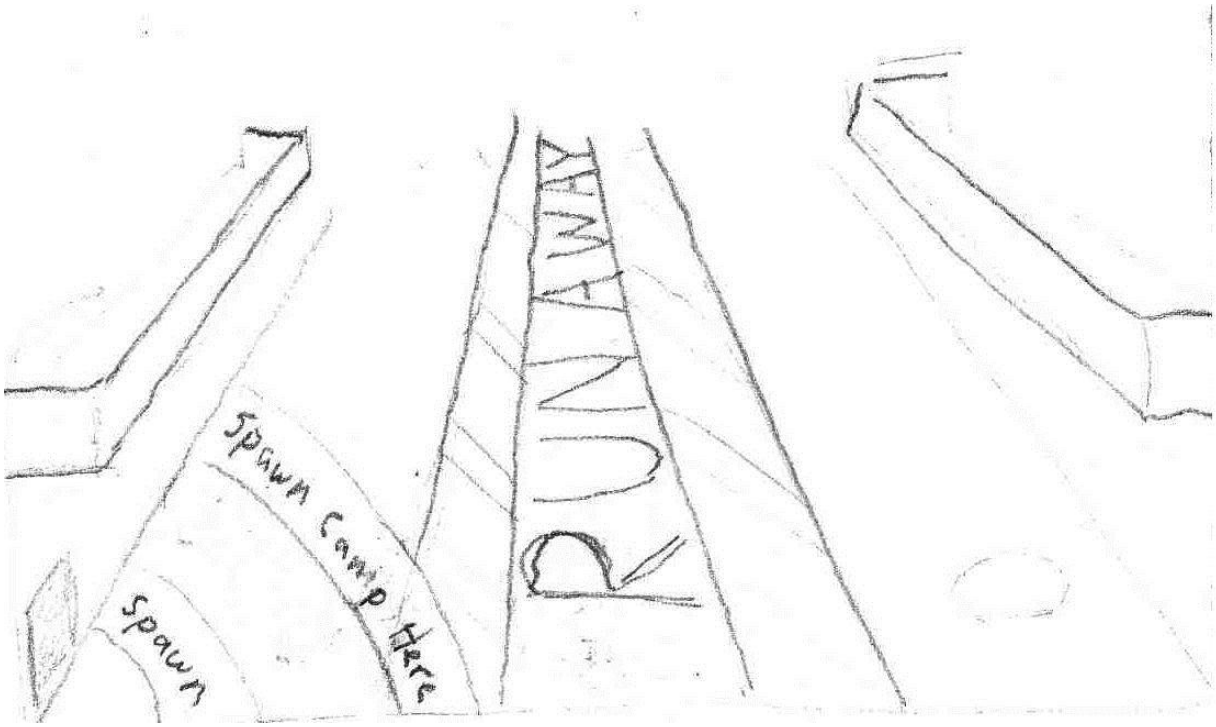
The Higher Learning Arena

Environment

The floor in this room is covered in colorful, rubberized material that is high traction and easy to clean. The walls are constructed from stone with pads on their bottom sections. Various educational areas are arranged around it, including a target for practicing the knock back modifier and decoys. It has a variety of layouts emphasizing the different situations on an FPS (cover, high ground, spawn camping, etc.).

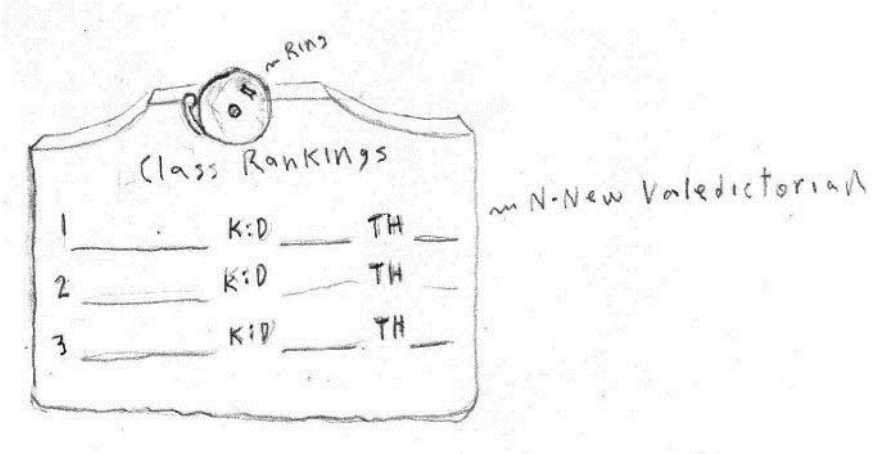
Learning Opportunity Decals

- *Description:* Colorful decals that identify locations. Meant to teach students about combat. Examples include: "Spawn Camping Area, Sniper Spot, Run Away, Spawn Area, Taunt, Photo Op"
- *Quality:* Low, 2-50/piece.
- *Use*: Placed on the floors and walls around the arena.
- *Importance*: Gameplay (1). Art (5). Can be simplified in number and ornateness.



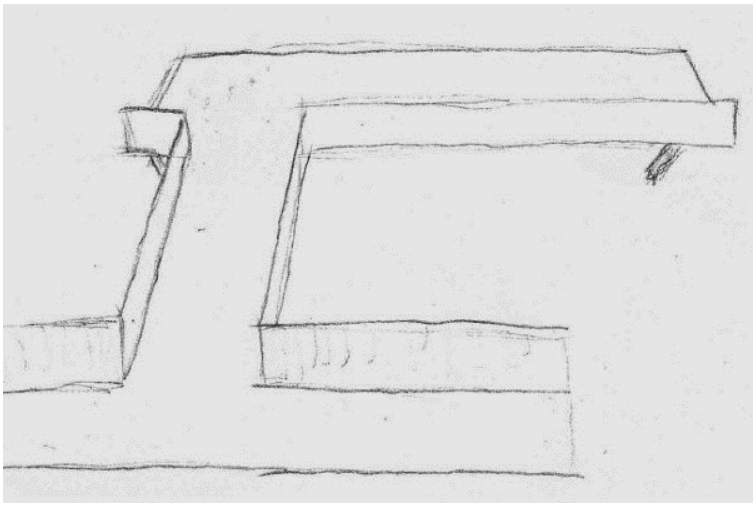
Grade Scoreboard

- *Description:* Wall mounted scoreboard depicting ranks for each student along with kill: death ratio and number of trap triggers. Bell mounted on the top rings every 30s and then scoreboard announces "N-n-new Valedictorian" if there is a new leader.
- *Quality:* Low, 130.
- *Use*: Placed on a non-covered wall at a height to place between the 1st and 2nd level.
- *Importance*: Gameplay (2). Art (5). Can be stripped of functionality.



Cat Walks

- *Description:* Metal cat walks of about 8 feet in width. Secured to the wall on the bottom with bolted L-shaped pieces of metal. Colored dark grey.
- *Quality:* Medium, 400/piece.
- *Use*: Attached to the walls and between other cat walks. Two floors vertical.
- *Importance*: Gameplay (5). Art (3). Can play with the railing details.



The Classrooms

Environment

Stone walls and floors. Items are colored blue or red based on which team the student is a part of. Some parts of the classroom are mock battlefield environments. There are six total, stacked with three per floor on two floors.

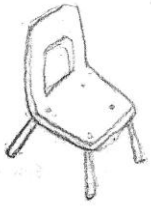
Chalkboard

- *Description:* Chalkboards depicting the name of the subject taught in the classroom. Roughly 4' x 12'. Classes include:
    - Team Combat: Defense

    - Team Combat: Offense
    - The Importance of High Ground
    - Spawn Camping: Do's and Don’ts
    - Our Violent History of Warfare: Tips and Tricks
    - Don't Shoot the Camera: Combat as Performance Entertainment
- *Quality:* Low, 70.
- *Use*: Attached to one classroom wall.
- *Importance:* Gameplay (1). Art (5). Number of classrooms can be reduced.

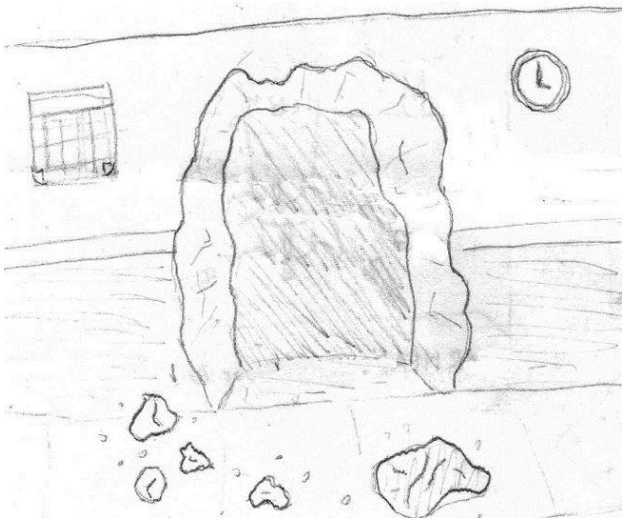
Chairs

- *Description:* Standard mass produced seating. Folding wooden chairs.
- *Quality:* Medium, 150.
- *Use*: Placed around the edges of the classroom.
- *Importance:* Gameplay (2). Art (3). Folding mechanism can be simplified.



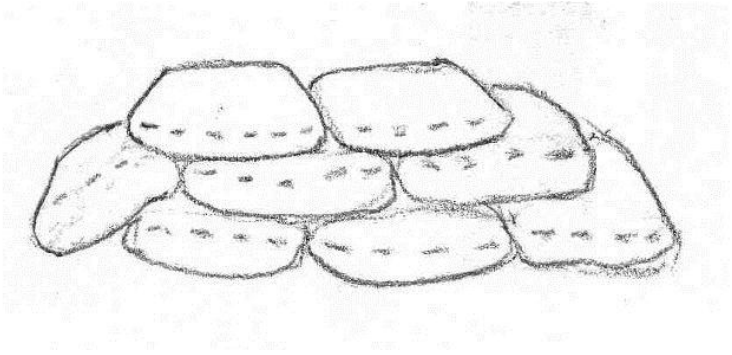
Rubble

- *Description:* Small and medium sized rocks clustered around the holes that were punched in the class room walls.
- *Quality:* Low, 50/piece.
- *Use*: Placed around the edges of the classroom.
- *Importance*: Gameplay (1). Art (3). Number of different types of rocks can be reduced.



Sandbag Wall

- *Description:* Sandbags stacked in a wall to provide protection against bullets. Tall enough to block the bottom 2/3 of a person and wide enough to cover a small tunnel in width.
- *Quality:* Low-Medium, 50/piece.
- *Use*: Placed blocking the exit from the tunnel connecting the Team Combat: Offense class to the Team Combat: Defense class.
- *Importance*: Gameplay (4). Art (4). Cannot be simplified.



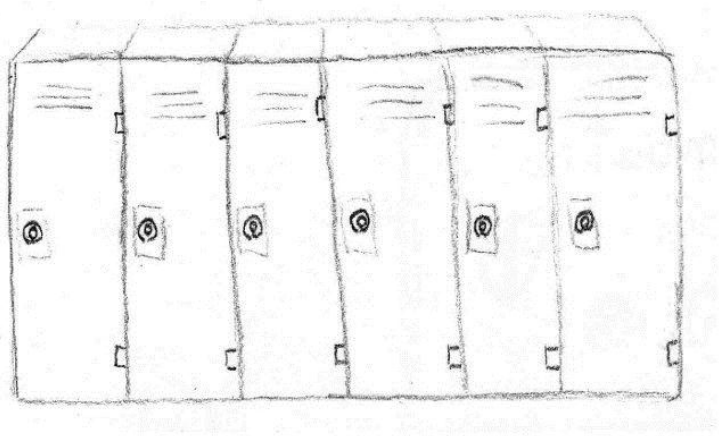
The Classroom Hallway

Environment

Brick halls with tile floors. Posters and lockers line the walls.

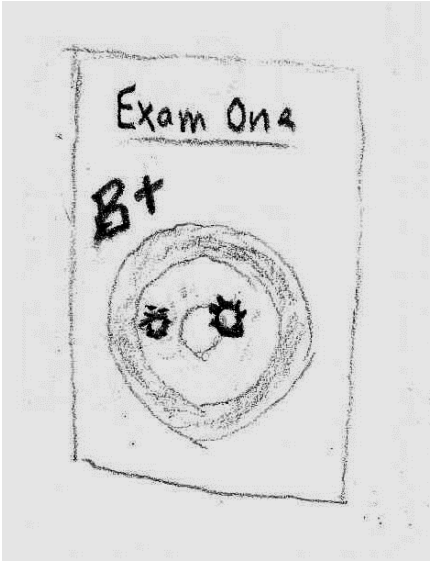
Lockers

- *Description:* Gun-metal gray lockers with combination locks. Six to a bank. 5' tall.
- *Quality:* Low, 40.
- *Use*: Placed on one side of the hallway to act as cover.
- *Importance:* Gameplay (4). Art (4). Cannot be simplified.



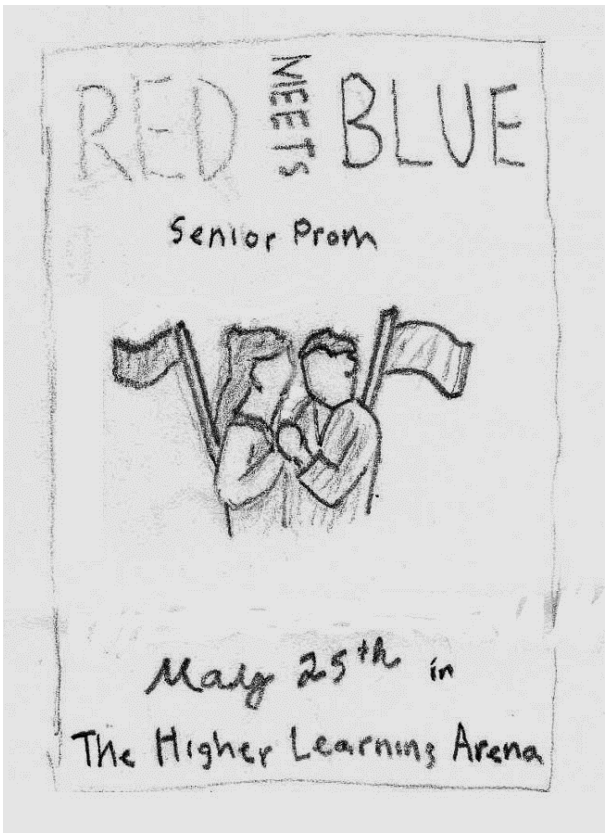
Graded Shooting Targets

- *Description:* Black and white, cartoonish shooting target graded with a B+.
- *Quality:* Low, 2-10.
- *Use*: Posted on a locker.
- *Importance:* Gameplay (1). Art (3). Cannot be simplified.



School Dance Poster

- *Description:* Poster advertising for the School Dance: "Red meets Blue: Senior Prom"
- *Quality:* Low, 2-10.
- *Use*: Placed in between the lockers.
- *Importance:* Gameplay (1). Art (3). Cannot be simplified.


RED MEETS BLUE
Senior Prom
May 25th in
The Higher Learning Arena

The Society for Historical Beatdowns Wing

[Note: This wing can optionally be closed off to reduce the size of the map / asset creation. Hidden / optional assets are labeled as (Optional) in their title.]

The Museum

Environment

This large hall typifies the personalities of the professors. It is a worn and crumbling edifice that has been patched over to give the impression of grandeur. Holes in the walls have been covered over. The architectural style is classical, and focuses on whites, reds, and golds.

Classical Greek Airlock

- *Description:* An ancient Greek style entrance. A doorway, flanked by two columns, with a triangular upper piece. The doorway itself is composed of two sliding stone doors that are pressurized as an airlock. Labeled with "AIRLOCK" (it was "restored," filling in where four missing letters were). The air lock explains some unusually well preserved artifacts.
- *Quality:* Medium, 300.
- *Use*: Functions as a sealed entrance to the museum if it is to be closed.
- *Importance*: Gameplay (2). Art (4). Cannot be simplified.

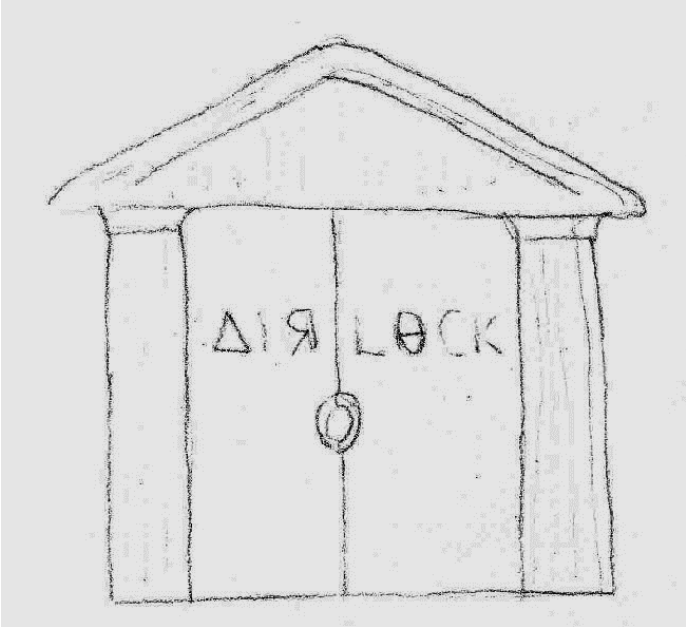
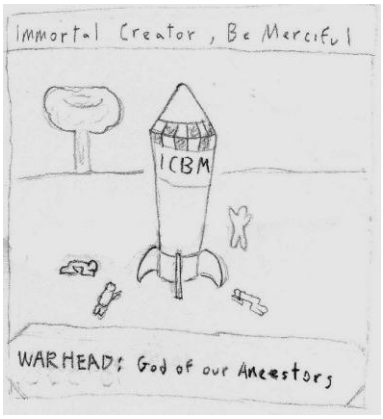


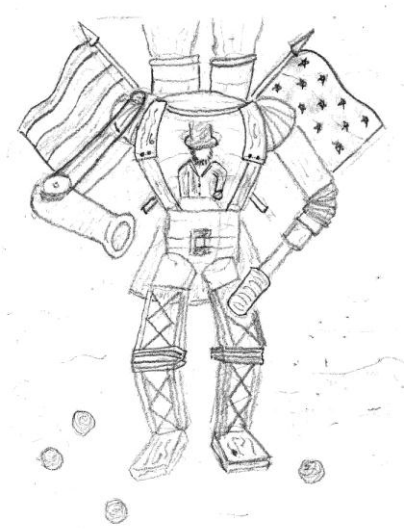
Exhibit Posters

- *Description:* Posters showing off the various exhibits inside the museum. One for each of the exhibits listed below: The Cannon, The Bowmerang, and The Gods of the Past.
- *Quality:* Low, 2-10.
- *Use*: Hung on the walls and doors around the museum.
- *Importance*: Gameplay (1). Art (4). Detail in the poster drawings can be reduced.



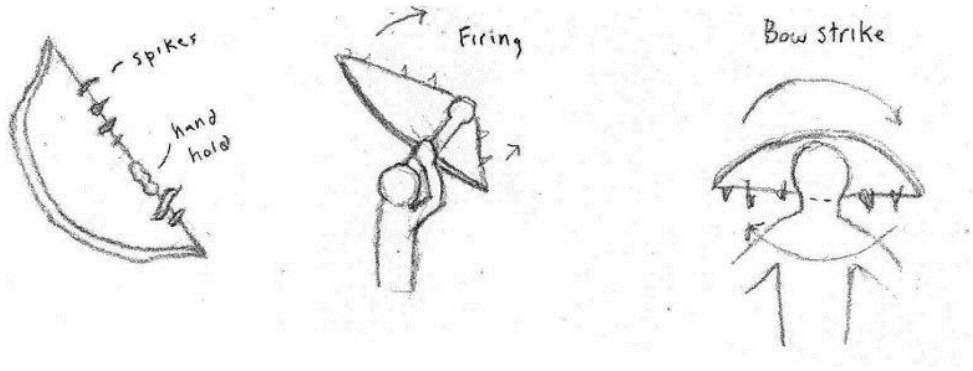
Cannon-wielding Exo-Skeleton (Optional)

- *Description:* A large, iron and wooden exo-skeleton. Dating back to the Civil War. Wields a cannon on one arm and a ramrod on the other. A misconception of how cannons are used.
- *Quality:* Medium, 2000-3000.
- *Use*: Placed in the museum.
- *Importance*: Gameplay (4). Art (4). Can be simplified, but must be its component parts must be recognizable (Civil War and the cannon).



Bowmerang Diorama (Optional)

- *Description:* A life-size diorama cordoned off by velvet rope. Depicts two men: the first is drawing back the bowmerang to shoot it, and the second is the soon to be victim. In between, semi-transparent bowmerangs are hung to depict the flight path. The bowmerang is a bow shaped piece of wood with its draw string covered in glass and spikes.
- *Quality:* Medium, 100 for bow + character.
- *Use*: Placed in the museum.
- *Importance*: Gameplay (3). Art (4). The mannequins can be simplified.

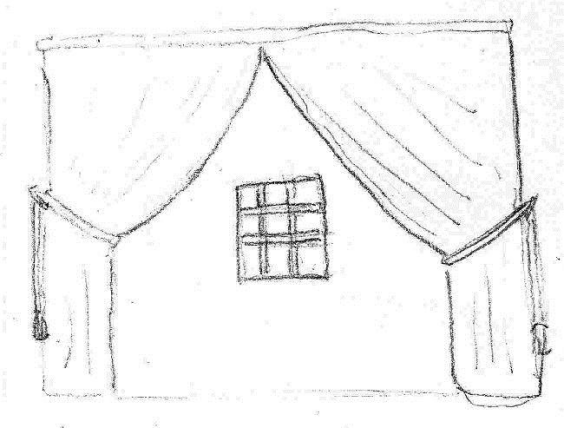


ICBM Diorama (Optional)

- *Description:* A larger-than-life ICBM surrounded by people worshipping/groveling. The ICBM is labeled with ICBM and Warhead. The backdrop shows a mushroom cloud.
- *Quality:* Medium.
- *Use*: Placed in the museum.
- *Importance*: Gameplay (4). Art (4). The number of worshippers can be reduced.

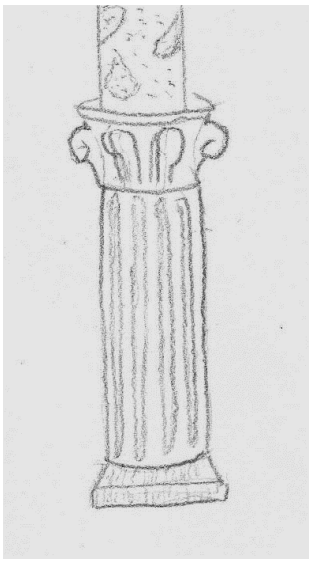
Drapery (Optional)

- *Description:* Drapery hung on the windows of the museum. It is made from a rich material like velvet or silk. The windows are small and almost prison-like, which is typical for a fort.
- *Quality:* Medium.
- *Use*: Hung on the windows around the museum. Cover defects and holes in the walls.
- *Importance*: Gameplay (1). Art (4). The draping of the fabric can be reduced to flatness.



Columns (Optional)

- *Description:* Plaster, Corinthian columns erected around the stone supports of the original fort. They only stretch up 2/3 of the original supports.
- *Quality:* Medium, 700/piece.
- *Use*: Supporting the ceiling.
- *Importance:* Gameplay (5). Art (4). Fluting detail can be reduced.



Impossible Possibilities Wing

Ground Level – Cubicles

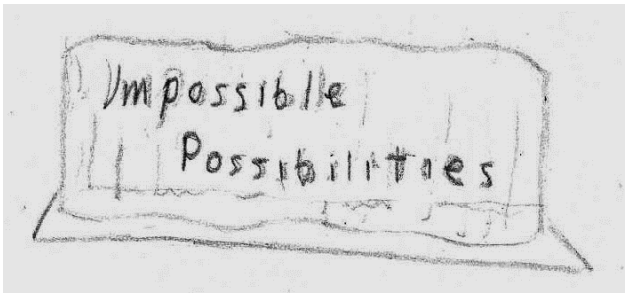
Environment

A tidy corporate environment. Well kept, well lit, and bland. Features fancy metallic "sculpture" hung on the windowed ceiling.

Impossible Possibilities Sign

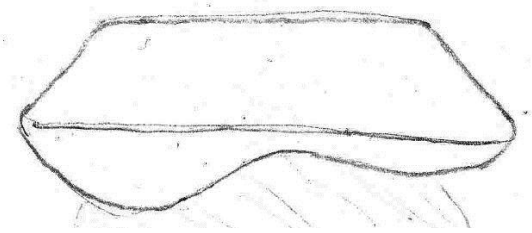
- *Description:* Futuristic looking sign displaying the Impossible Possibilities logo. It is holographic and floating in the air behind the welcome desk.
- *Quality:* Medium, 700.
- *Use*: Floating behind the welcome desk. Meant to be impressive.

- *Importance*: Gameplay (2). Art (4). Holographic effect can be reduced to a normal sign constructed of stainless steel.



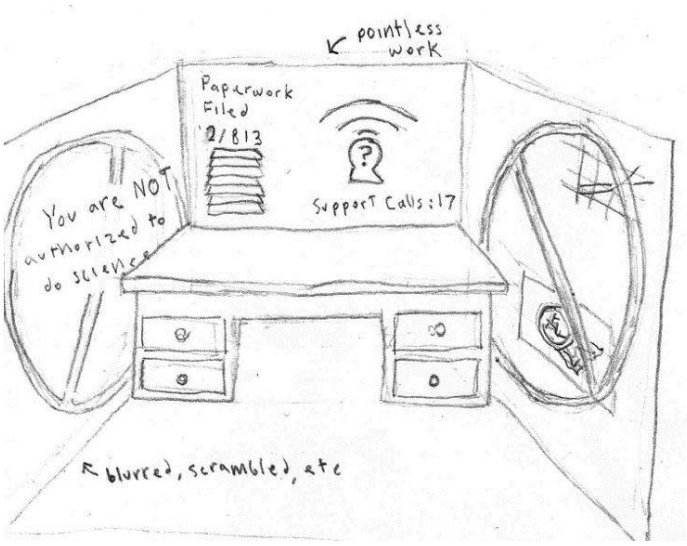
Welcome Desk

- *Description:* Made from glass. Wide but not bulky. Hovers above the ground.
- *Quality:* Low-Medium, 200.
- *Use*: Placed facing the entrance to this wing.
- *Importance*: Gameplay (3). Art (3). Can be simplified in shape.



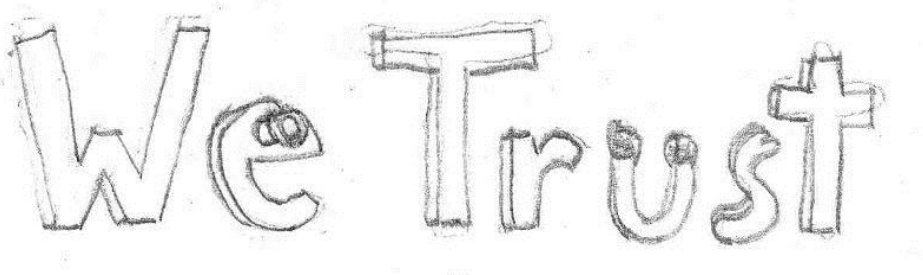
Cubicle

- *Description:* Six foot by six foot cell. Contains desk and chair. All four walls are information displays showing busy work (support calls, paperwork). Hacked to display impossibly good values.
- *Quality:* Medium, 200/piece.
- *Use*: Cover most of the upper level.
- *Importance*: Gameplay (5). Art (3). Textures can be simplified.



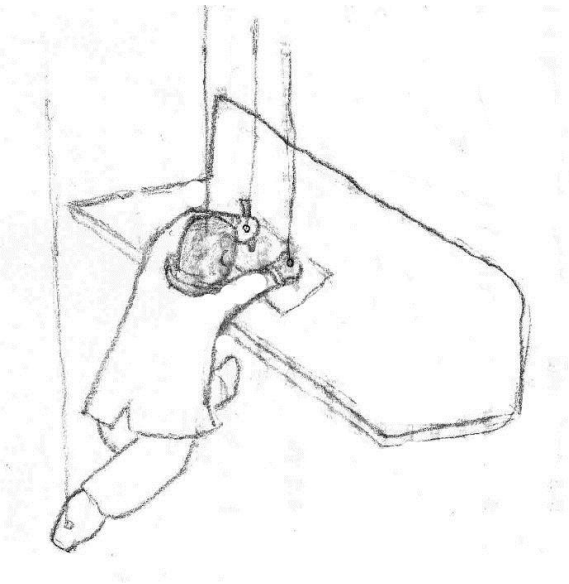
We Trust Our Employees Sign

- *Description:* A large corporate sign with the words "We Trust Our Employees." Set into the letters are microphones and video cameras.
- *Quality:* Low, 40-50.
- *Use*: Placed along one wall, facing all of the cubicle entrances.
- *Importance*: Gameplay (1). Art (4). Number of devices in the letters can be reduced.



Mannequin

- *Description:* Crude mannequins stolen from the ACF store and dressed in lab coats.
- *Quality:* Medium.
- *Use*: Placed in various cubicles to create the illusion of work.
- *Importance*: Gameplay (1). Art (3). Cannot be simplified.



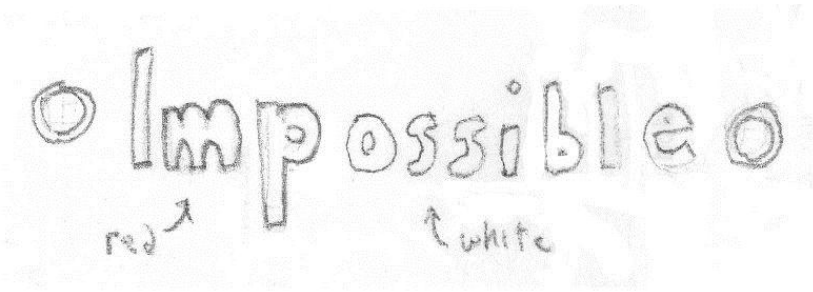
Basement Level – Lab

Environment

A large, subterranean area that is catacomb-like. Its walls are charred and deformed by explosions. In some places, the outlines of test subjects can be seen. It has two entrances: one from above in the office space area and one from a hall exiting the Academy wing. There are still various skeleton and torture devices around.

Suits Warning Sign

- *Description:* A white neon sign saying "Impossible Possibilities." The "Possibilities" part is burnt out. It displays "Possible" most of the time, but switches over to "Impossible" when the suits arrive. Flanked by red lights that also flash when the suits come by.
- *Quality:* Low, 2-10.
- *Use*: Placed in a high visibility spot to provide ample warning.
- *Importance:* Gameplay (1). Art (4). Functionality can be removed.



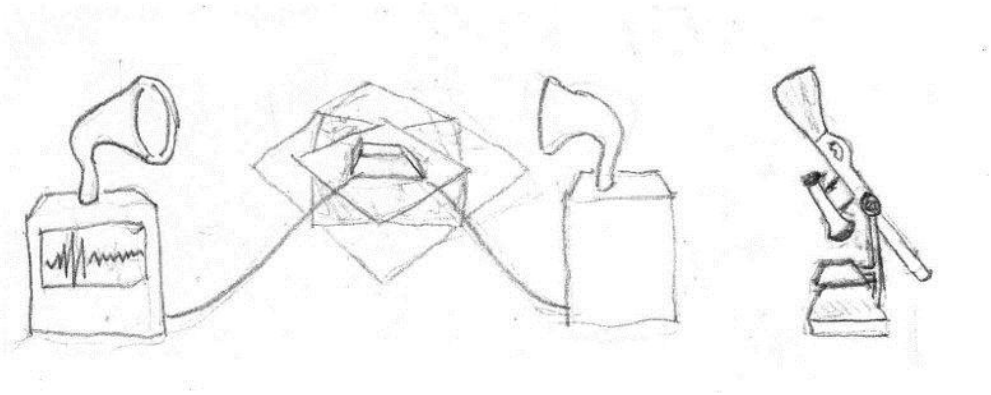
Entrance Warning Light and Sign

- *Description:* A red light embedded in a sign saying "Warning: Somebody is coming." The light activates whenever a player is standing in any of the inbound hallways.
- *Quality:* Low, 40.
- *Use*: Placed near the entrances to the Basement level.
- *Importance*: Gameplay (3). Art (3). Cannot be simplified.

Experimental Apparatuses

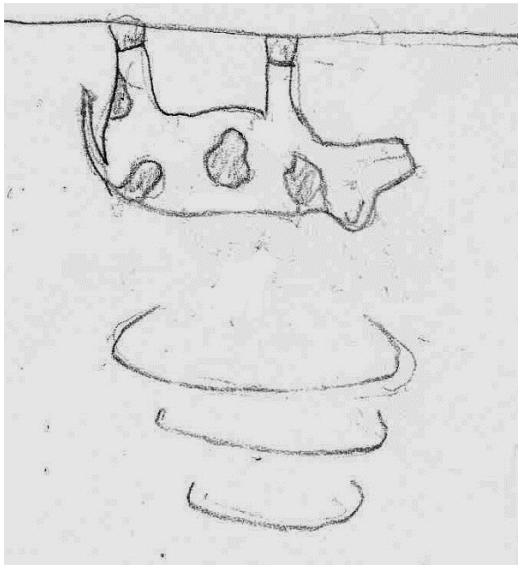
- *Description:* Modifiers that are being tested and devices for testing them. They apply their effects to the immediate area around them. When the suits alarm goes off, they fold up into the wall.
- *Quality:* Medium.
- *Use*: Placed around the room.

- *Importance:* Gameplay (3). Art (4). Can be reduced in number and complexity.



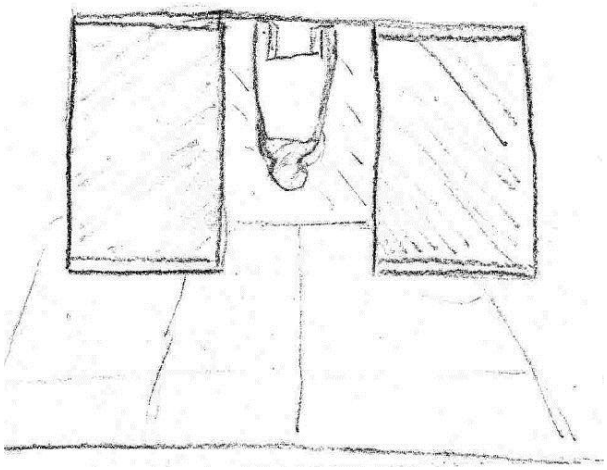
Test Subjects

- *Description:* Various living and non-living objects that are used to test the modifiers. Examples are: Cows and Mannequins.
- *Quality:* Medium.
- *Use*: Placed in the experimental areas.
- *Importance:* Gameplay (2). Art (3). Can be replaced with simpler test subjects.



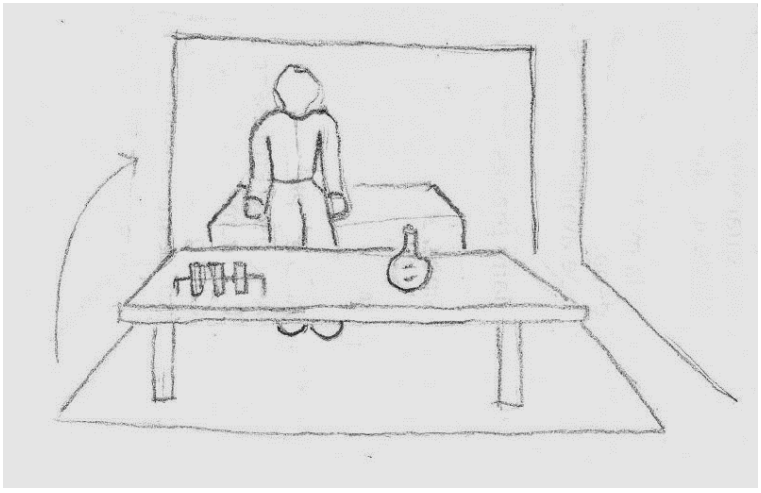
Inverted Cubicle

- *Description:* A cubicle environment attached upside-down to the ceiling. It is placed in a gravity inversion area to test the effects of the modifier.
- *Quality:* Medium, 200/piece.
- *Use*: Placed in the room on the ceiling.
- *Uses Content From*: **Ground Level – Cubicles** "Cubicle."
- *Importance:* Gameplay (4). Art (4). Cannot be simplified.



Hide-able Lab Bench

- *Description:* A laboratory bench with various electronic parts placed on its service. A curved back chair by it with a scientist sitting on it. Folds up into the wall.
- *Quality:* Medium, 100.
- *Use*: Placed along one wall, facing all of the cubicle entrances.
- *Importance*: Gameplay (4). Art (3). Ornaments on bench surface can be removed.



Animations

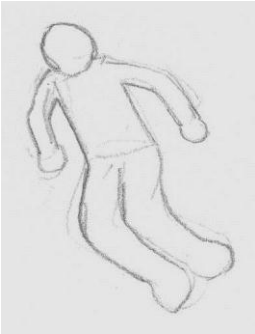
Character Animations

Walk

Action: A medium speed jogging motion. As each foot prepares to strike the ground, the other foot has already left it.

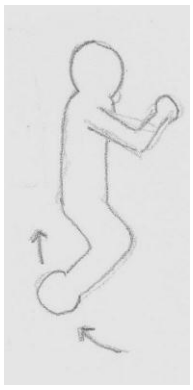
Die

Action: The player collapses to the ground with arms splayed out and knees bent.



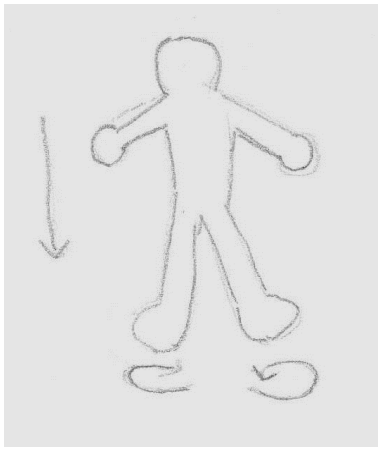
Jump

Action: The player jumps upwards bending the knees and raising the upper legs.



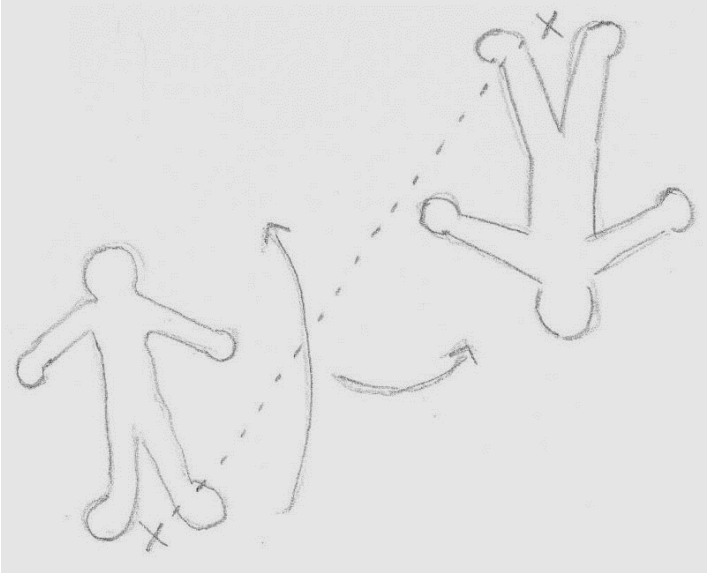
Fall

Action: The player model extends the legs and moves them in small circles.



Invert

Action: The player begins moving in the opposite direction of the current gravity vector while the model rotates 180 degrees about the X and Y axes.



Grow

Action: Scales the player model up and slows the walk cycle animation.

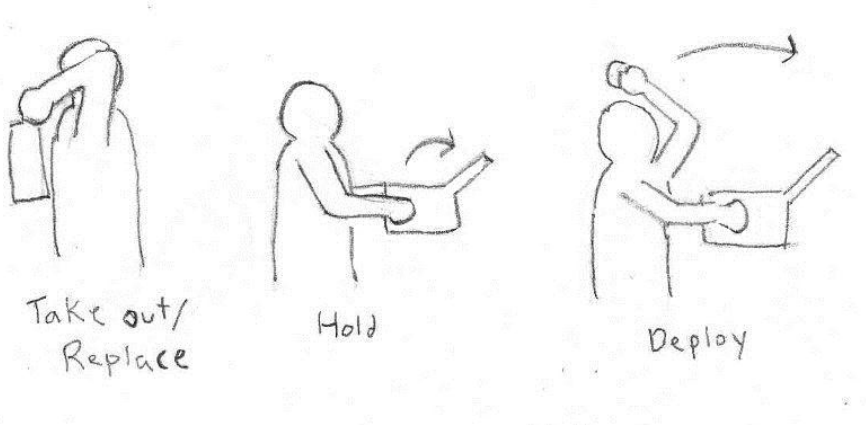
Shrink

Action: Scales the player model down and speeds up the walk cycle animation.

Weapon Animations

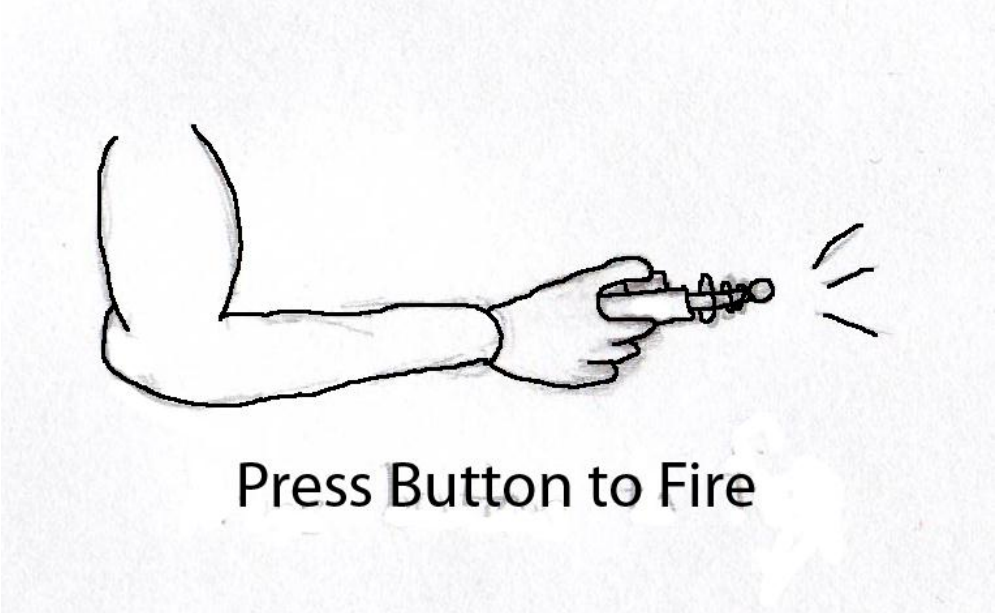
Modifier Launcher

- **Hold:** Two-handed. Clutches the box on both sides.
- **Deploy:** Holds the box on the right, reaches in with the left, grabs a modifier and flings it by straightening the arm.
- **Take out/Replace:** Reaches over the shoulders towards the back with both hands.



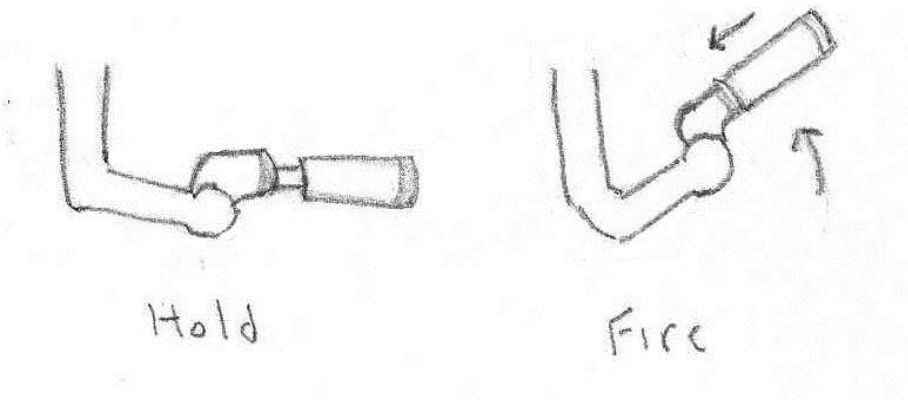
Nullifier Weapon

- **Hold:** One-handed. Hand grips the remote shape from the side.
- **Deploy:** Presses the button on the surface with the holding hand.
- **Take out/Replace:** Reaches down toward the legs.



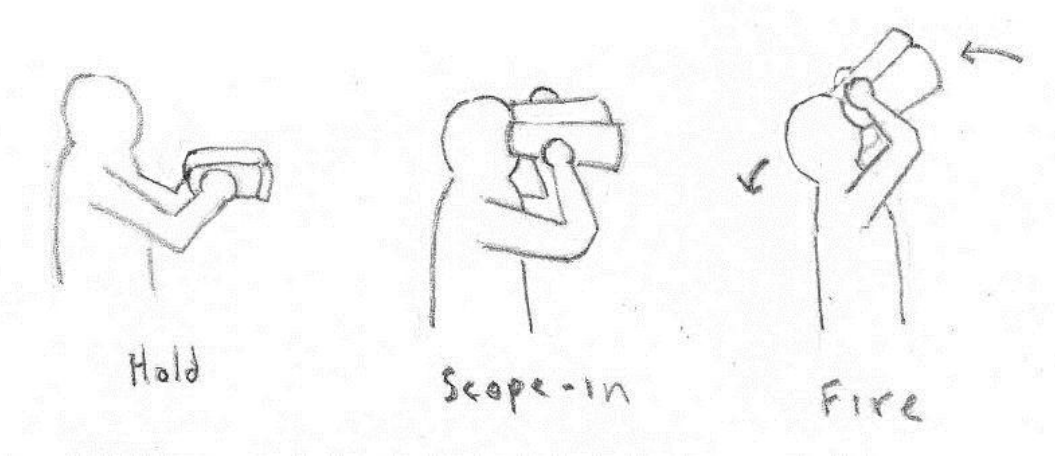
Pistol

- **Hold:** One-handed. Bobs with the player's hand as they move.
- **Fire:** Small amount of recoil for a normal shot. Larger recoil for a charged shot. The barrel moves toward the hand when a shot is fired and moves back out shortly after the shot is fired.
- **Take out/Replace:** Reach down toward legs.



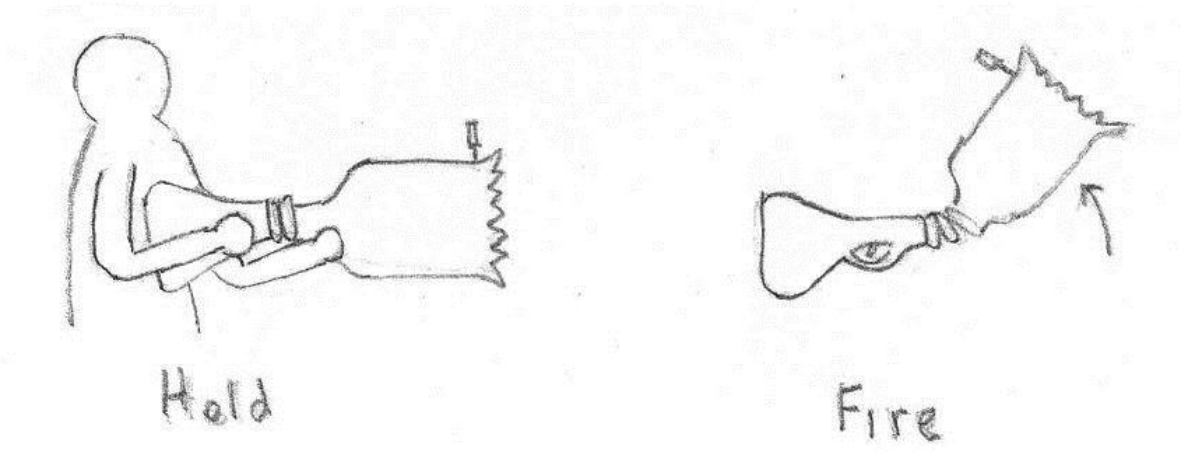
Sniper Rifle

- **Hold:** Two-handed. Hands grip either side of the binocular sniper. Held at chest level.
- **Scope-in:** Move the binoculars to the eyes.
- **Fire:** Significant recoil. When scoped-in, the head whips backwards. When not scoped-in, the binocular sniper strikes the chest.
- **Take out/Replace:** Reach down toward legs.



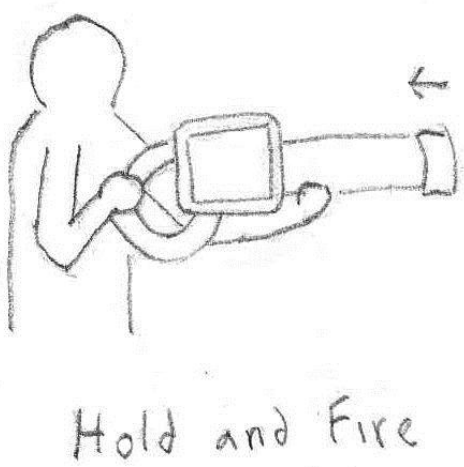
Explosive Projectile Weapon

- **Hold:** Two-handed. Bobs as the player moves.
- **Fire:** Heavy recoil. Snaps the forward half of the gun upwards at the joint. Rebounds back into place after completing the upward motion.
- **Take out/Replace:** Reach down toward legs.



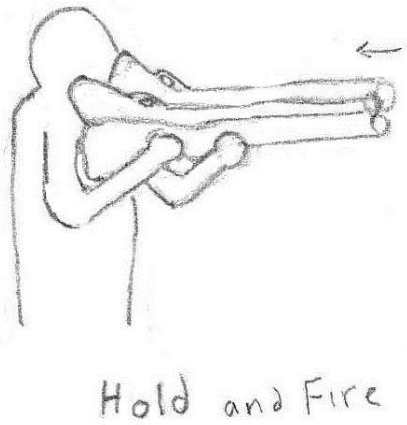
Assault Rifle

- **Hold:** Two-handed. Bobs with the player's hands as they move.
- **Fire:** Very limited recoil. Moves slightly up and toward the player as it's shot.
- **Take out/Replace:** Reach down toward legs.



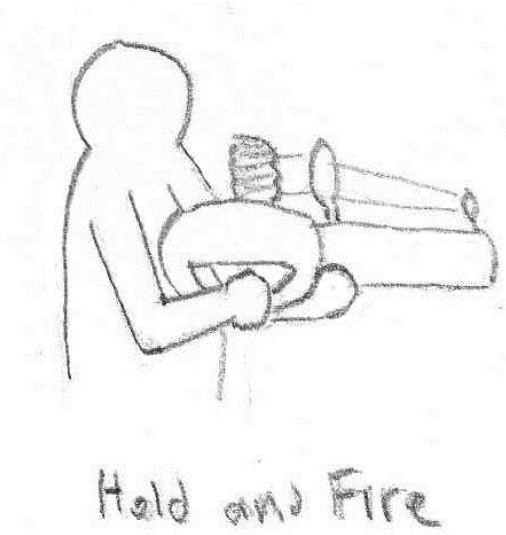
Shotgun

- **Hold:** Two-handed. Bobs with the player's hands as they move.
- **Fire:** Heavy recoil moving the barrel significantly up and back. The three component guns reload sequentially.
- **Take out/Replace:** Reach down toward legs.



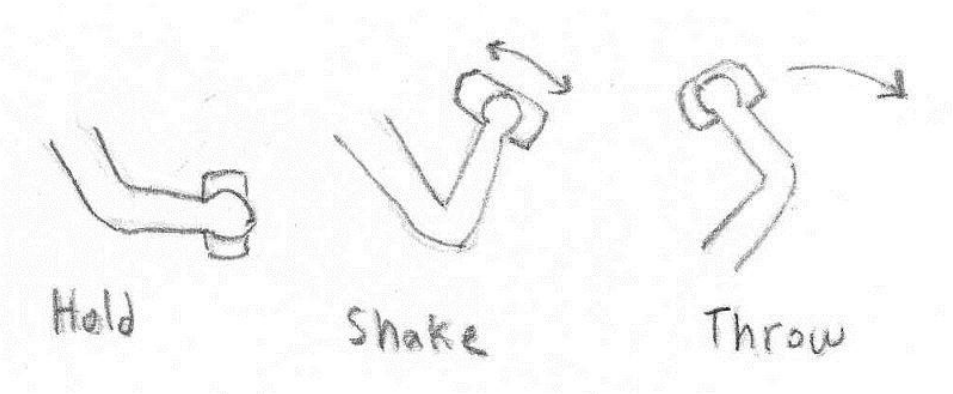
Damage over Time Weapon

- **Hold:** Two-handed.
- **Fire:** No recoil. A beam projects through the lenses and to the target.
- **Take out/Replace:** Reach down toward legs.



Grenade

- **Hold:** Single-handed. Bobs with the player's hand as they move.
- **Shake:** Bends at the elbow up and down rapidly in a shaking motion.
- **Throw:** Extends the elbow and rotates the shoulder forward to throw the grenade.



Technical Design Document

Summary

This document is used to provide a detailed view of the technical aspects of the development for Trigger Happy the game. All aspects of both the design of the game objects as well as the underlying game engine have been detail in the pages below. Since the design choices made for the engine are tangential to the usual design choices for engines, some confusion might result if the reader does not start from the beginning with the system architecture description.

Document Scope

This document is intended to be used in the development and implementation of the Trigger Happy game. The document will detail the architecture of the multi-threaded engine, the core components, and gameplay elements that are to be used in the development cycle. It is useful background reading for anyone involved in management or oversight of the Trigger Happy game.

Development Technology

Graphical Platform	NVidia 9600 GT (x2)
System Hardware	Intel Core2 Quad 2.66GHz
Content Pipeline Tools	Autodesk Maya
	Apple Logic
	Adobe Photoshop CS4
Development Tools	Visual Studio 2008
	SVN (gdd.unfuddle.com)
	Trac
Operating System(s)	Windows Vista
	Windows 7
External Libraries	Lua v5.1
	XACT
	DirectX v10
	DirectInput v8

Production Technology Requirements

Hardware	NVidia 9 Series graphics card or greater Radeon HD2900 graphics card or greater
	Intel Pentium D 3GHz or Higher
Operating System(s)	Windows Vista
	Windows 7
External Libraries	DirectX v10 or greater
	DirectInput v8
	Lua v5.1

Engine

Details of the internal details of the Singularity engine, its architecture, and objects.

System Architecture

At the core of Trigger Happy's development is the design of a new engine known as Singularity. Singularity combines a task-based threading model, popularized by the Intel's Threaded Building Blocks, and a component-based entity architecture. By combining these two techniques Singularity is able to fully utilize multi-core processors while minimizing the usual overhead associated with multi-core development. In the context of multi-core processors, the combination of these two architectures is very powerful and offers a world of possibilities not available to most game engines.

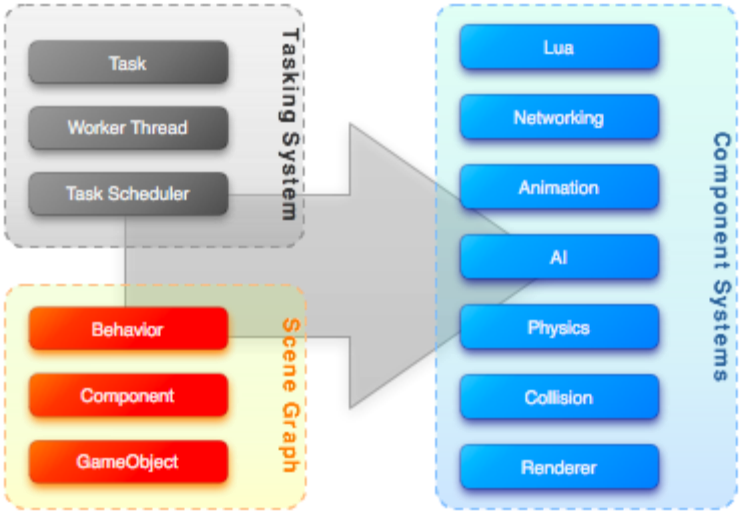


Figure 39: Singularity System Architecture Overview

The engine has split into multiple subsystems that consist of the tasking system, scene management, and the engine components. Both the tasking and scene management systems are core to the engine framework and provide the means of executing or managing the engine components. Components are functional extensions to the engine and help to supplement the needs of game development via physics, animation, or other game systems.

Multi-core

With the advent of multi-core processors in the past five years, game engines have begun to move towards parallel execution architectures. Intel's release of the Threaded Building Blocks (TBB) library was a huge advancement allowing developers to easily interface with the system's cores. Though the core of design is similar to Intel's Threaded Building Blocks, the library has been rewritten to allow more control over the scheduling and memory footprint of the tasks.

Scheduling for TBB is a "first come first serve" model, allowing tasks to execute quickly but with no ordering or absolute scheduling. Singularity's design attempts to extend the concept of TBB to enable

task ordering and allows tasks to dictate how quickly they should be executed. In order to implement these additions, a new scheduling system and task usage tracking was added.

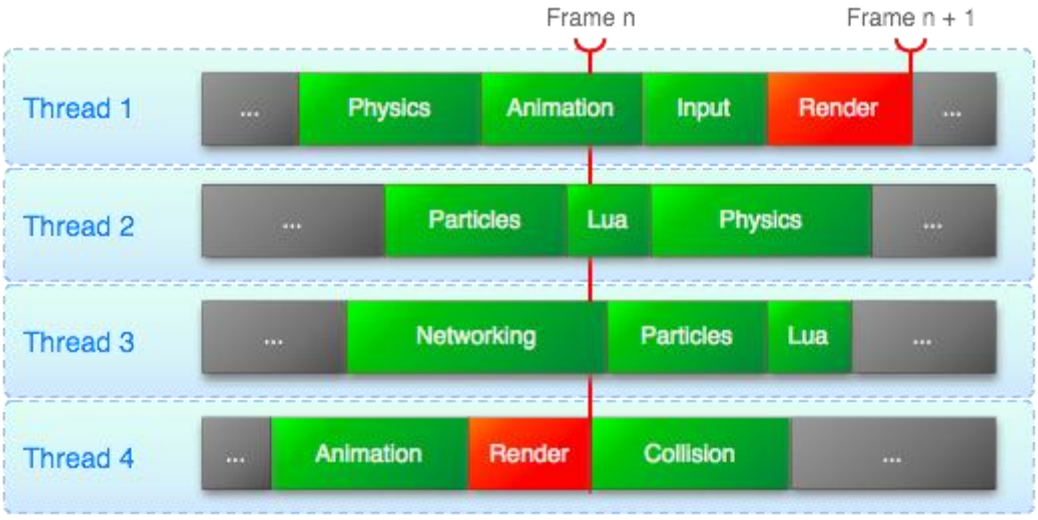


Figure 40: Threading Task Scheduling Diagrams

To further move the tasking system into the gaming environment, the addition of dependency ordering was added. This feature allows chaining of tasks through a dependency chain. This prevents the render task from running before other systems such as physics or the particles. If the tasks did not have this feature, then tasks would be unpredictable and the rendered output would not match the state of the system properly.

Componentization

One of the core requirements for the Singularity engine was to allow quick development with an easy to build system. In order to accomplish such a task, many different techniques were researched. Each technique had its strengths and weaknesses; however, the component-based entity approach gave the engine the flexibility, testability, and compatibility to work within the multi-core threading model.

A component architecture is based on the separation of functionality into individual components. Instead of the traditional object hierarchy, an object is created as a collection of components; the sum of which represents all of the functional needs of the entity. Each component runs only within the scope of its functionality removing much of the data dependency of hierarchical models.

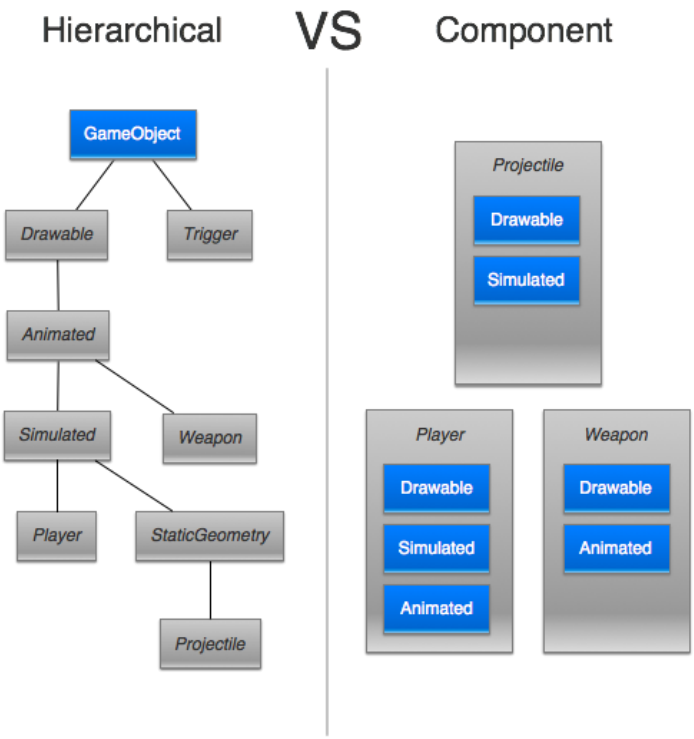


Figure 41: Hierarchal Architecture vs. Component-based Architecture

The advantages of the component architecture allow us to build independent component systems to handle the usual game functionality. Components all use to minimize the amount of overhead associated with multi-threaded systems. Due to their independent nature, components can minimize the use of data synchronization constructs. With very little data sharing between threads there is are only a small number of cases that require any form of locking mechanism. Unlike most threading models that are only data or functionally separated, the threading model and component architecture integration allow Singularity to utilize a threading model based on data and functional division.

In Singularity, all game objects are created from a single type. However, unlike the hierarchical method, we do not derive from the class. Instead we build up the class by adding components and behaviors. Components add functionality such as animation or model rendering; behaviors are scriptable and allow developers to extend the functionality of the game object.

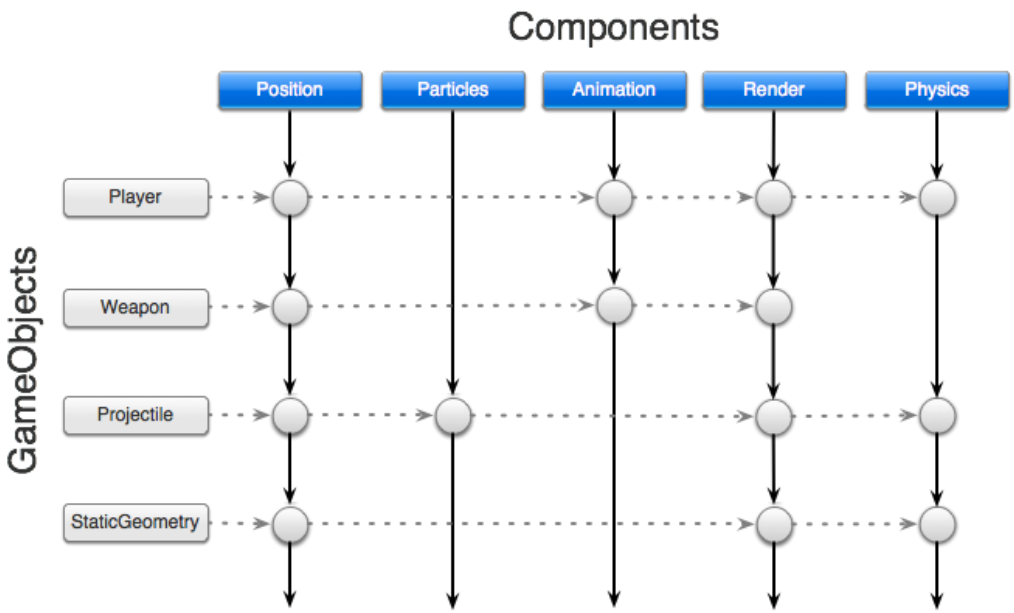


Figure 42: Object Composition Using Components

When the components are being executed, they are controlled by a centralized extension singleton. The purpose of this object is to split up the task into smaller more manageable tasks, each with a subset of the initial set of components. By running components in parallel instead of focusing on the game object, we can build a threading model where very few locks are needed to maintain stability. With a small number of locks, we can utilize the tasking system and the multi-core system to its fullest. What this means for the developer is more time to perform AI, physics, or other lengthy execution systems without having to worry about frame rate.

Tasking System

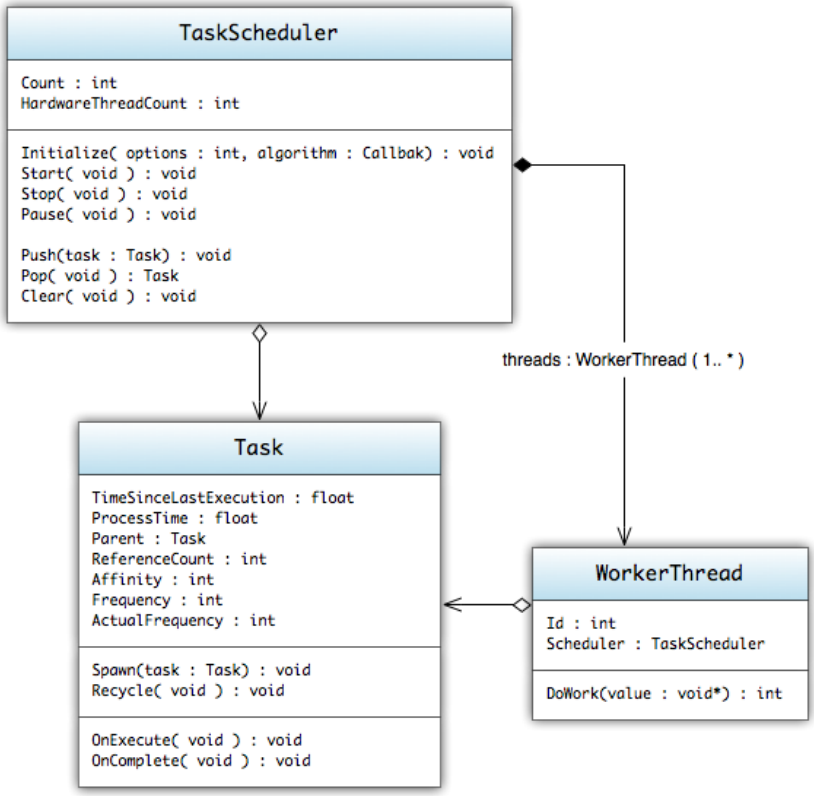


Figure 43: Task System UML Diagram

The tasking system is a hybrid of the two techniques used in the Intel Smoke demo; both of which utilize the Intel Threaded Building Blocks library. To minimize the systems size and execution overhead, the tasking system uses a custom developed systems architecture that closely mirrors the functionality of the Intel Threaded Building Block. Instead of managing tasks on each worker thread, the adjusted system architecture manages a centralized queue of tasks. Also, due to the system component based architecture, the standard game loop has been replaced with a dependency driven task execution pipeline.

TaskScheduler

The task scheduler is the core of the tasking system; its primary function is to manage the scheduling of tasks according to a weighting algorithm. The scheduler maintains an internal queue of the currently assigned tasks and is structured as a dequeue. When a task is requested the scheduler returns the next task according to the weighting algorithm.

Weighting Algorithm

The rules used to determine the what task is executed next

- The task whose last child was completed by this thread.
- The task whose frequency deadline has been reached or will fail if another task is executed
- A task with the highest affinity
- A task with the shortest process time

- **Data**
  - Count : *int*
    - Number of *Tasks* currently on the scheduling queue
  - HardwareThreadCount : *int*
    - Number of cores/processors that are available on the current system

- **Operations**
  - Initialize(options : *int,* algorithm *: WeightingAlgorithmCallback*) : *void*
    - Initializes the task scheduler using the options. Allows the system to run in single or multi-threaded mode as well as scheduling behaviors.
  - Start(void) : void
    - Starts the scheduling system and spools up the appropriate *WorkerThread*
  - Stop(void) : void
    - Stops the scheduling system and shuts down the *WorkerThreads*
  - Pause(void) : void
    - Pauses the scheduling system and puts all of the *WorkerThread* into sleep mode
  - Push(task : *Task*) : void
    - Pushes the task onto the scheduling queue.
  - Pop(void) : Task
    - Pops the next task from the scheduling queue. Priority is determined by task frequency, last execution time, affinity, and process time. Ideally tasks that have a frequency set will be moved to the start of the queue the closer they get to their deadline.
  - Clear(void) : void
    - Removes all of the currently queued tasks from the scheduling queue.

WorkerThread

The *WorkerThread* is the execution thread needed to process the queue tasks. Depending on the options specified in the initialization of the *TaskScheduler* the number of *WorkerThreads* can range from one up to the number of cores the system has.

- **Data**
  - Id : *int*
    - Identifier of the *WorkerThread* as assigned by the *TaskScheduler*
  - Scheduler : *TaskScheduler*
    - The scheduler which is currently handling the task management.

- **Operations**
  - DoWork(value : *void**) : int
    - The main execution loop for the *WorkerThread*. This is where all of the *Task* Execution happens as well as the idling when no more tasks are present.

Task

The task is the primary development concept of the tasking system. Tasks are the unitized functional calls that allow the threading model to execute dependent on the processor core, thus allowing multicore programming without having to maintain the overhead of thread management.

The task offers a set of functions that allow the developer to manage child and parent relationships.

- **Data**
  - TimeSinceLastExecution : *float*
    - Measurement, in seconds, of the elapsed time since the last execution of the task; if this is the initial run then the return value is 0.
  - ProcessTime : *float*
    - Measurement, in seconds, of the amount of time the task requires to complete. This value is averaged over the execution of the task.
  - Parent : *Task*
    - If the task was spawned from another task, this will be the value of the task's originator. Tasks that have not been spawned from a task will not have a parent assigned.
  - Scheduler : *TaskScheduler*
    - The scheduler which is currently handling the task management.
  - ReferenceCount : *int*
    - The number of tasks that have been spawned from this task. Before the task can complete every sub-task must complete, thus bringing the reference count to 0.
  - Affinity : *int*
    - The task's affinity within the current task scheduler. The lower the number the higher the priority in the scheduler and the task will run sooner.
  - Frequency : *float*
    - This defines, for the system, how often the task should be run. If no frequency is set, the system will treat the task as a filler when dealing with the scheduling algorithm.
  - ActualFrequency : *float*
    - Measurement of the tasks actual execution frequency. Due to the dynamic scheduling and process times associated with tasks, the actual execution frequency may not be the same as the defined frequency.

- **Operations**
  - Spawn(task : *Task*) : *void*
    - Spawns a child task associated with the current task. Completion of the parent task is dependent on the completion of all of the spawned child tasks. The reference count of the task will increase by one for each spawned task.
  - Recycle(*void*) : *void*
    - Commands the task to push itself back onto the queue once it has finished execution and has been completed (all of its children have completed).
  - OnExecute(*void*) : *void*
    - This function is called when the task has been assigned to a *Worker Thread* and is executed. When deriving from *Task*, the primary function that is adjusted to accommodate the functional needs.
  - OnComplete(*void*) : *void*
    - This function is called when the task and all its children have completed and the task is finally pulled from the task queue. If the task has been recycled it will be reset and pushed back onto the scheduling queue.

Scene System

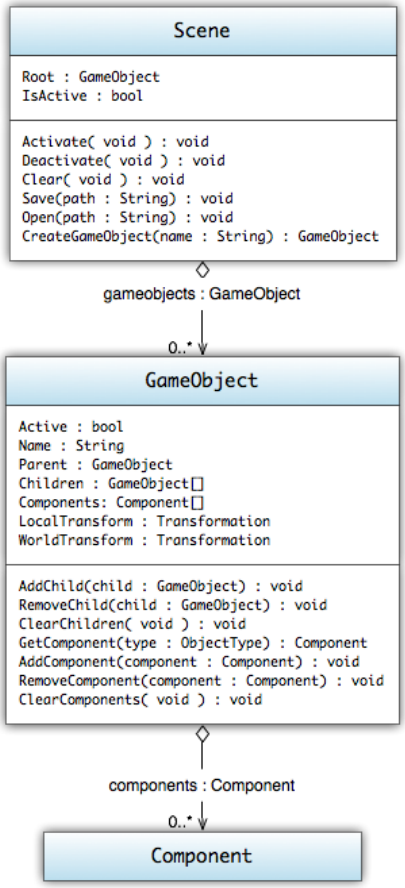


Figure 44: Scene System Diagram

The scene system is the core of the engine and manages all of the objects that are to be placed within Trigger Happy. Basic scene culling and object partitioning is handled within this system via the SceneGraph system. The system allows customization of the storage and partitioning of the *GameObjects* by the inclusion of user definable *SceneGraph* classes.

GameObject

The *GameObject* are containers for all of the engine's components. All of the objects in Trigger Happy are compositions of other *GameObjects* and *Components*. *GameObjects* do not add any characteristics to the game by themselves; instead, they manage the attached *Components* which implement the actual functionality.

- **Data**
  - Active : *Boolean*
    - Details whether the *GameObject* is active within the *SceneGraph*
  - Name : *String*
    - The name of the object
  - Parent : *GameObject*
    - The parent of the game object
  - Children : *GameObject[]*

        - The children of the game object
    - Components : *Component[]*
        - The list of all the attached components
    - LocalTransform : *Transformation*
        - The local transformation of the *GameObject*, this transform describes the *GameObjects* position, rotation and scale relative to the parent *GameObject*
    - WorldTransform : *Transformation*
        - The world transformation of the *GameObject*, this transform describes the *GameObjects* absolute position, rotation and scale in the world.

- **Operations**
    - AddChild(child : *GameObject*) : *void*
        - Adds a child *GameObject* to current *GameObject*; thus setting the child's parent to the current *GameObject*
    - RemoveChild(child : *GameObject*) : *void*
        - Removes the child *GameObject* from the current *GameObject*; if the child is not found then nothing is removed
    - ClearChildren(*void*) : *void*
        - Removes all of the child *GameObjects* from the current *GameObject*
    - GetComponent(type : *ObjectType*) : *Component*
        - Retrieves the first instance of a component that matches the *ObjectType*
    - AddComponent(component : *Component*) : *void*
        - Adds a *Component* to the current *GameObject*
    - RemoveComponent(component : *Component*) : *void*
        - Removes the component from the current *GameObject*; if the component is not found then nothing is removed
    - ClearComponents(*void*) : *void*
        - removes all of the components from the current *GameObject*

Component

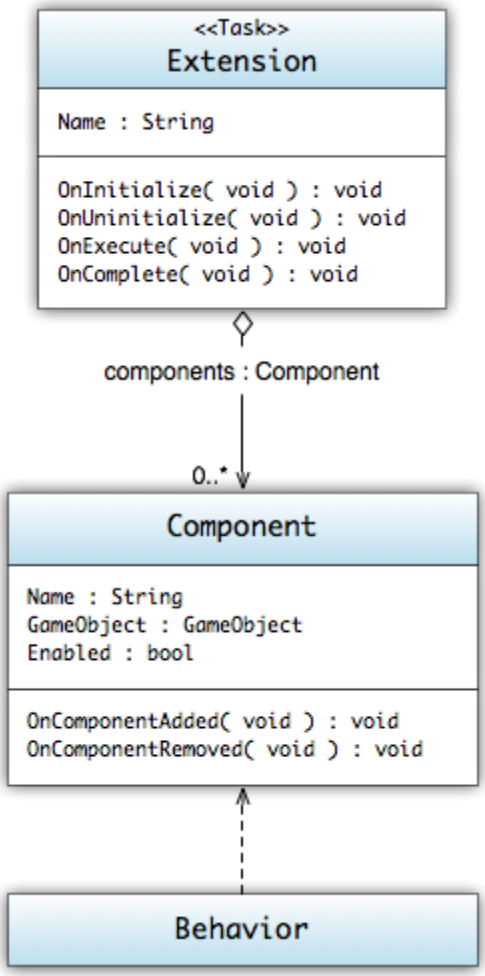


Figure 45: Component UML Diagram

Components represent all of the functional additions that can be attached to *GameObjects*; examples include rendering, collisions, physics, etc. See the Components section (section 6) for the current list of engine components that are available to the designers and the game development.

- **Data**
  - Name : *String*
    - Contains the user-defined name of the *Component*
  - GameObject : *GameObject*
    - The GameObject that the *Component* is attached to
  - Enabled : *Boolean*
    - Describes whether or not the *Component* will be executed

- **Operations**
  - OnComponentAdded (*void*) : *void*
    - Called when the *Component* is added to a *GameObject*
  - OnComponentRemoved (*void*) : *void*
    - Called when the *Component* is removed from a *GameObject*

Behavior

The *Behavior* class is nothing more than a simple distinction between functional *Components* and a completely callback based *Component.*

Extension

Extensions are the managers of components and execute the overall behavior that the custom *Component* is meant to embody. *Extensions* are also *Tasks* and as such are able to take advantage of the multicore tasking system. By utilizing the multicore tasking system, *Extensions* are able to partition their *Components* and execute each subset in parallel; greatly increasing the execution throughput of the system.

- **Data**
  - Name : *String*
    - Contains the user-defined name assigned to the *Extension*

- **Operations**
  - OnInitialize(*void*) : *void*
    - Called when the first *Extension* of this type is initially loaded into the scheduler
  - OnUninitialize (*void*) : *void*
    - Called when the last *Extension* of this type is unloaded from the scheduler
  - OnExecute(*void*) : *void*
    - Inherited from Task
  - OnComplete (*void*) : *void*
    - Inherited from Task

Scene

The Scene is the organizational construct that maintains and orders the *GameObjects*. Culling and other spacial operations will also be performed using the power of the *SceneGraph*. Finally the management of object locking is maintained by the *SceneGraph* in order to prevent systems from causing data collisions through the manipulation of the *GameObjects* within the *SceneGraph*.

- **Data**
  - Root : *GameObject*
    - Root *GameObject* that all other *GameObjects* are relative to. This is normally used as the base coordinate system for the game.
- **Operations**
  - BeginUpdate(*void*) : *void*
    - Sets up a differed rebuild of the *SceneGraph* tree structure. This is used to prevent a large adjustment of *GameObjects* from propagating on every change; speeding up the system significantly.
  - EndUpdate(force : *Boolean*) : *void*
    - Finalizes the differed update of the *SceneGraph* tree structure. If *force* is enabled, then the tree will rebuild whether or not the tree needs it or not.
  - Update(force : *Boolean*) : *void*
    - Instantly rebuilds the *SceneGraph* tree structure. If *force* is enabled, then the tree will rebuild whether or not the tree needs it or not.

Components

The engine design up to this point has been about the framework and subsystems needed to manage and execute the components. Components are the tools by which designers and developers can take the engine to create a game. As new functionality is needed, new components can be added further extending the abilities of the engine.

Component	Description
Rendering	Component system used to render images to the graphics device or screen. Maintains all the information about the graphics hardware and builds in some core components for building scenes.
Input	Component system for tracking and recording input data.
Collision	Component system used to track and find collisions between objects.
Physics	Component system the enables physic based simulations on game objects. This system is used in addition to the collision system to allow general in-game physics simulations.
Animation	Component system that controls and updates the meshes to enable character or object animations. Gives the animator a set of functional tools for controlling animation sequences and transitions.
Networking	Component system that manages and synchronizes multiple game clients to enable multi-player action.
Lua	Component system that allows lua scripting to be executed and interact with game objects.
Particle	Component system for displaying and managing particle systems.
Audio	Component system for playing and managing audio. The audio components allow for 3D positional audio and dynamic audio content playback.
GUI	Component system that attaches to the camera component to draw 2D GUI components.
Content	Component tools for loading and building game objects based off of attribute encoded models.
In-Game	Components that are needed to extend the core engine components in order to fill gaps needed for the development of Trigger Happy.

Rendering Subsystem

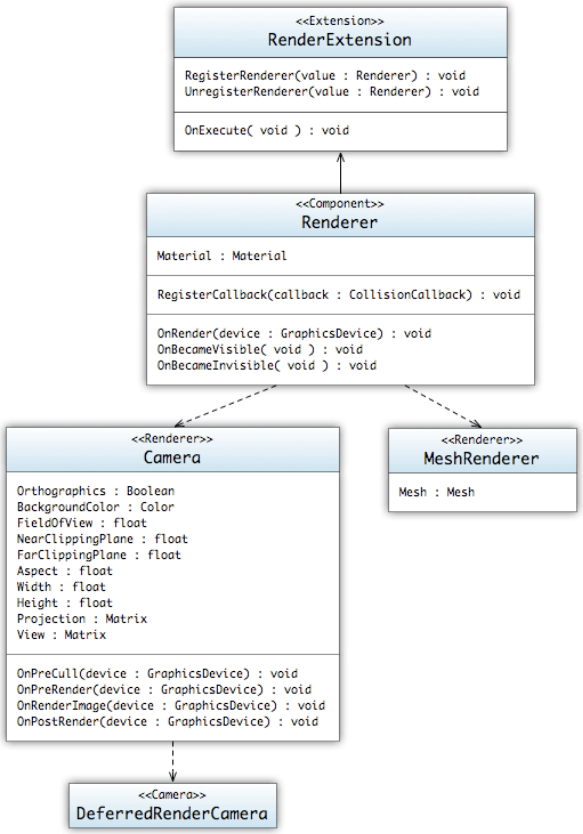


Figure 46: Render UML Diagram

The graphics renderer is the component interface through which calls to the graphics hardware are made. It manages all aspects of the rendering and is the sole means through which one may request objects to be drawn to the screen. Since the Singularity engine is component based, the rendering pipeline is implemented as a collection of different components. Handling of all drawing calls is dealt with through the use of both the *Camera* component and the *MeshRenderer*. Though both are *Renderers*, each component handles different aspects of the rendering pipeline. The *Camera* primarily handles the setup and processing of the viewport and resulted images from rendering. The *MeshRenderer* handles the object based rendering.

Mesh

Represents a collection of vertices and indices that represent a 3D/2D object.

- **Data**
  - BoundingVolume : *BoundingVolume*
    - A bounding object that contains all of the mesh's polygons.
  - Vertices : *VertexBuffer*
    - The mesh's *VertexBuffer*
  - Indices : *IndexBuffer*
    - The mesh's *IndexBuffers*

- **Operations**

- Clear(*void*) : *void*
  - Clears out the vertex and index buffers of the mesh resulting in an empty mesh.
- RecalculateBounds(*void*) : *void*
  - Recalculates the bounding volume of the mesh.
- SetVertices(declaration : *VertexDeclaration*, data : *void[]*, length : *int*) : *void*
  - Creates or modifies the *VertexBuffer* with the provided data.
- SetIndices(data : *int[]*, length : *int*) : *void*
  - Creates or modifies the *IndexBuffer* with the provided data.

Material

Materials are the core of the rending pipeline. Materials represent effects and as such control how objects are drawn and represented on the screen.

- **Data**
  - PassCount : *int*
    - The number of passes associated with this material.
- **Operations**
  - SetVariable(key : *String*, value : *Object*) : *void*
    - Sets the materials parameter named *key* with the value provided.

GraphicsDevice

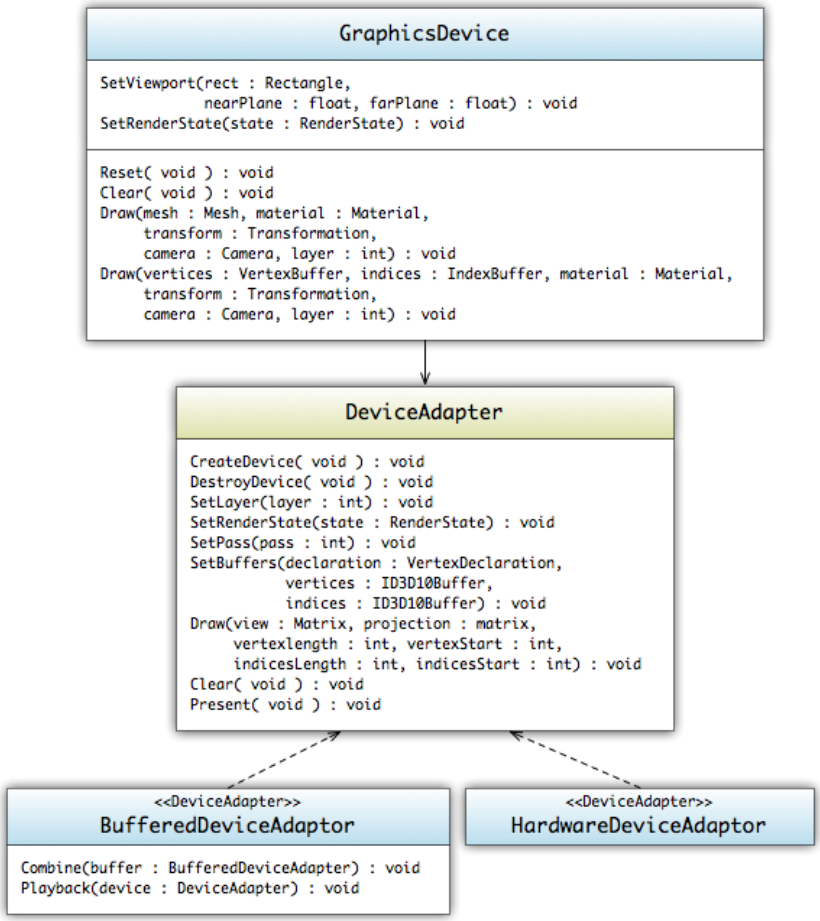


Figure 47: Graphics UML Diagram

The *GraphicsDevice* used in Singularity has a different structure than most graphic devices in other engines. Most engines spend approximately 50-80% of their time in the render state. Since the majority of the execution time will be used on rendering, it is in our best interest to spread this execution across the cores. However, due to the nature of the graphics hardware, we are unable make render calls across threads without locking each call. With Singularity's multi-core system, this slowdown would cause great hiccups in the execution process. To alleviate this issue, the graphics device has two primary modes. One mode is a direct line to the underlying hardware device, the other is a recording system that both optimizes the render calls and is thread safe. The recording system allows the render of all of the visible objects using the power of the multi-threaded system without having to block between render calls. Once the recording has been completed, the *RenderExtension* spools up the *HardwareRendererExtension* which will playback the recorded rendering calls.

Figure 48: Example of Threaded Rendering

DeviceAdapter

The *DeviceAdapter* is a base class for the graphics rendering pipeline. The class's main purpose to provide an interface layer between the objects and the device. In Singularity's case, there are two *DeviceAdapter* provided with the engine. One is the *HardwareDeviceAdapter* which is a DirectX 10 wrapper for the device calls. The other *DeviceAdapter* is the *BufferedDeviceAdapter*, this adapter is a call recorder, and its main purpose is to allow the recording of draw calls for a playback at a later time.

- **Operations**
  - CreateDevice(*void*) : *void*
    - Creates and initialize the *DeviceAdapter.*
  - DestroyDevice(*void*) : *void*
    - Destroys and releases and resources used by the *DeviceAdapter.*
  - SetLayer(layer : *int*) : *void*
    - Sets the layer that the *Draw* call will occur on; Layers are a form of draw ordering.
  - SetRenderState(state : *RenderState*) : *void*
    - Sets the current *RenderState* to the specified settings.
  - SetBuffers(declaration : VertexDeclaration, vertices : *ID3D10Buffer*, indices : *ID3D10Buffer*) : *void*
    - Sets the draw buffers that are to be used when drawing; the indices are not required to be able to draw. If this is the case, then all of the vertices buffer must contain all of the vertices in draw order to be able to draw.

- Draw(view : *Matrix*, projection : *Matrix*, vertexLength : *int*, vertexStart : *int*, indicesLength : *int*, indicesStart : *int*) : *void*
  - Draws the vertices/indices to the resources (screen/recorder) used.
- Clear(*void*) : *void*
  - Clears the resource (screen/recorder) buffer of the drawn objects.
- Present(*void*) : *void*
  - Finalizes the resource (screen/recorder) buffer and draws out the objects.

HardwareDeviceAdapter

The *HardwareDeviceAdapter* is just a wrapper around the DirectX 10 device calls. It provides an easier to use interface for the developers by abstracting away all of the device setup and management of render states.

BufferedDeviceAdapter

The *BufferedDeviceAdapter* is a recording class that takes all of the draw calls and stores them into a tree structure. Recording the draw calls allows us to enabled threaded rendering and provides the engine a means to organize and optimize the draw calls.

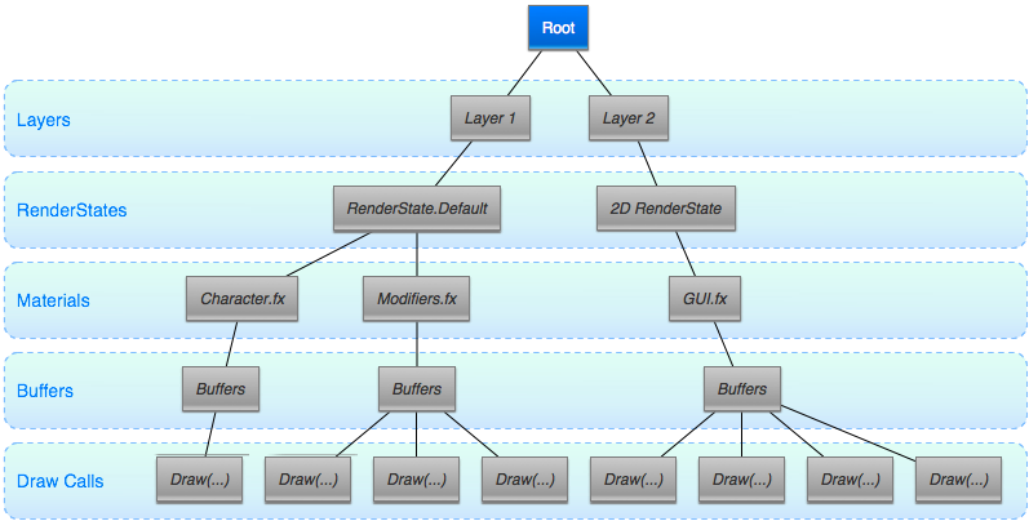


Figure 49: Example of the recording tree structure used in BufferedDeviceAdapter

Each instance of the device manages their own copy of the tree structure. When the final render task is completed, the trees are combined and played back using the *HardwareDeviceAdapter*. The technique allows the other threads to start working on other tasks and builds the *HardwareDeviceAdapter* a tree with an optimized render pipeline, speeding device rendering considerably.

- **Operations**
  - Combine(buffer : *BufferedDeviceAdapter*) : *void*
    - Combines the two *BufferedDeviceAdapter* Recordings into the parent. Due to the tree and depth nature of the recording tree, combination is a quick union of the recording trees.
  - Playback(device : *DeviceAdapter*) : *void*

        - Plays back the recorded render calls to the device. Most cases this will be the *HardwareDeviceAdapter;* however, if a file recording system was created it could be passed into *BufferedDeviceAdapter* to enable recording to a file.

Renderer

The *Renderer* is the base *Component* that is used to render images to the screen. Though the class itself is abstract, the *Renderer* is the only means of a *Component* to be given the objects to be able to draw to the hardware device. At the current iteration, the only classes that implement *Renderer* is the *Camera* and *MeshRenderer*. The *Camera* is used to setup the viewport and manage both post-render and pre-render processes, where as the *MeshRenderer* is used to draw out the objects to the device.

- **Data**
    - Material : *Material*
        - The *Material* that will be used to render out objects.

- **Operations**
    - OnRender(device : *GraphicsDevice*) : *void*
        - Called when the *RenderExtension* is executed to render the screen and the object is not culled. The *GraphicsDevice* provided hides whether the rendering will be buffered or directly drawn to the screen.
    - OnBecameVisible(*void*) : *void*
        - Called when the object was originally culled from rendering but has now been allowed to render.
    - OnBecameInvisible(*void*) : *void*
        - Called when the object was originally rendered but has now been culled from rendering.

MeshRenderer

The *MeshRenderer* is the primary *Component* for rendering a Mesh to the screen.

- **Data**
    - Mesh: *Mesh*
        - The *Mesh* that will be used to render out objects.

- **Operations**
    - OnRender(device : *GraphicsDevice*) : *void*
        - Inherited from *Renderer; c*alled when the *RenderExtension* is executed to render the screen and the object is not culled.
    - OnBecameVisible(*void*) : *void*
        - Inherited from *Renderer;* called when the object was originally culled from rendering but has now been allowed to render.
    - OnBecameInvisible(*void*) : *void*
        - Inherited from *Renderer;* called when the object was originally rendered but has now been culled from rendering.

Camera

The *Camera* is a *Component* through which the player views the world. A screen space point is defined in pixels. The bottom-left of the screen is (0,0); the right-top is (*Width*, *Height*). A viewport

space point is normalized and relative to the camera. The bottom-left of the *Camera* is (0,0); the top-right is (1,1).

- **Data**
  - Orthographic: *Boolean*
    - Whether or not to use orthographic projection.
  - BackgroundColor : *Color*
    - The background color of the *Camera.*
  - FieldOfView : *float*
    - The field of view of the camera in degrees.
  - NearClippingPlane : *float*
    - The near clipping plane distance.
  - FarClippingPlane : *float*
    - The far clipping plane distance.
  - Aspect : *float*
    - The aspect ratio (width divided by height).
  - Width : *int*
    - The width of the camera in pixels
  - Height : *int*
    - The height of the camera in pixels
  - Projection : *Matrix*
    - Matrix that transforms from world space to camera space
  - View : *Matrix*
    - Matrix that transforms the model space to world space

- **Operations**
  - OnRender(device : *GraphicsDevice*) : *void*
    - Inherited from *Renderer; c*alled when the *RenderExtension* is executed.
  - OnPreCull(device : *GraphicsDevice*) : *void*
    - Called before a *Camera* culls the scene.
  - OnPreRender(device : *GraphicsDevice*) : *void*
    - Called before a *Camera* starts rendering the scene.
  - OnRenderImagel(device : *GraphicsDevice*) : *void*
    - Called after all rendering is complete to render image
  - OnPostRender(device : *GraphicsDevice*) : *void*
    - Called after a *Camera* has finished rendering the scene.

RendererExtension

The *RenderExtension* is the primary task used to render *Meshes* to the render screen. This extension spawns a multitude of subtasks to render out the *Meshes* and *Cameras* using the *BufferedDeviceAdapter*; once the tasks have completed, the *BufferedDeviceAdapters* are combined and played back using the *HardwareDeviceAdapter*.

Note: Most tasks will register themselves as parent dependencies of the RendererExtension, this makes the render task move to the end of the scheduling process.

- **Operations**
  - OnExecute(*void*) : *void*

- Inherited from *Task*; Spawns a multitude of sub-tasks to render the *Meshes* to the screen.

Input Subsystem

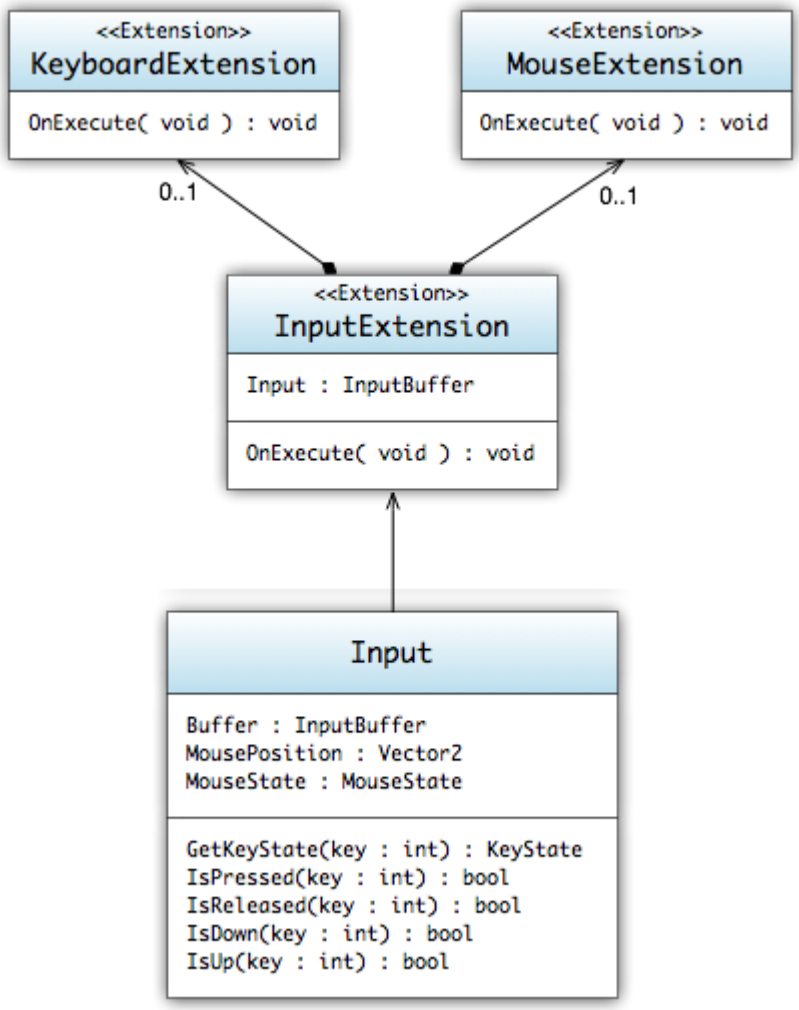


Figure 50: Input UML Diagram

The input system runs at approximately 30 Hz and gathers all of the inputs attached to the system. The system also provides intermediary calculations to provide easy interfaces for interacting with both the keyboard and mouse devices.

Note: Any tasks that require input from the user must register a dependency with the InputExtension.

InputBuffer

Manages and stores all of the input state.

- **Data**
  - Buffer : *InputBuffer*
    - The current buffered state of the input devices.
  - KeyStates : *KeyState[]*

        - The current states of all of the keys.
    - MousePosition : *Vector2*
        - The Current position of the mouse relative to the windows client area.
    - MouseState : *MouseState*
        - The current state of all of the mouse buttons.

Input

The *Input* class is a static thread-safe class that allows access to the device's state. The class is updated through the *InputExtension* and manages a double buffered system to prevent threading issues.

- **Data**
    - MousePosition : *Vector2*
        - Current position of the mouse relative to the window's client area.
    - MouseState : *MouseState*
        - Current state of the mouse; this constitutes which buttons are pressed, and where the scroll wheel is located
- **Operations**
    - GetKeyState(key : *int*) : *KeyState*
        - Gets the key's current state.
    - IsPressed(key : *int*) : *boolean*
        - Returns whether the key has been pressed; press constitutes an up event and then a subsequent down event.
    - IsReleased(key : *int*) : *boolean*
        - Returns whether the key has been released; release constitutes a down event and then a subsequent up event.
    - IsUp(key : *int*) : *boolean*
        - Returns whether the key is in the up state.
    - IsDown(key : *int*) : *boolean*
        - Returns whether the key is in the down state.

InputExtension

The *InputExtension* is the primary task used to gather the input state. This extension spawns both the *MouseExtension* and *KeyboardExtension* and manages the input state double buffer.

- **Data**
    - Input : *InputBuffer*
        - The current input state that has been gathered.
- **Operations**
    - OnExecute(*void*) : *void*
        - Inherited from *Task*; Spawns the two other tasks, *KeyboardExtension* and *MouseExtension*, and swaps buffers.

KeyboardExtension

The *KeyboardExtension* handles all of the calls to *DirectInput* keyboard device and updates the input state.

- **Operations**
    - OnExecute(*void*) : *void*

        - Inherited from *Task*; Queries and updates the input state.

MouseExtension

The *MouseExtension* handles all of the calls to DirectInput mouse device and updates the input state.n

- **Operations**
    - OnExecute(*void*) : *void*
        - Inherited from *Task*; Queries and updates the input state.

Collision Subsystem

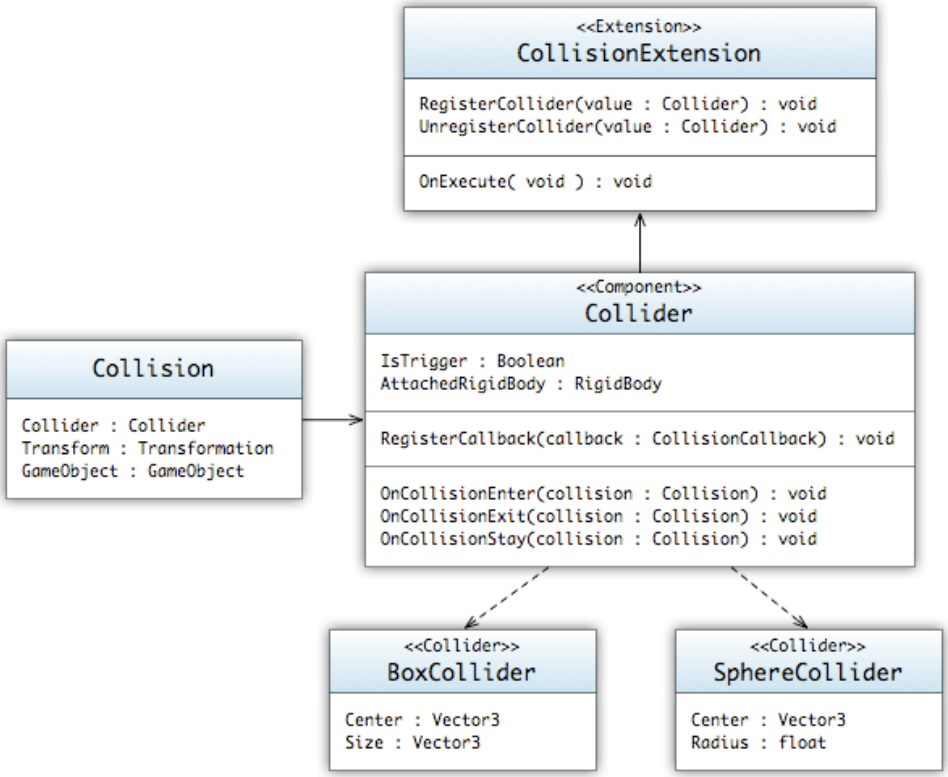


Figure 51: Collision UML Diagram

The collision system is a collection of customizable *Collision* components that allow *GameObjects* to interact. Due to the inherent response that most physics systems have, *Collision* components are loosely tied to the physics subsystem's *RigidBody* component. Without the *RigidBody* attached the *Collider* has no means of influencing the *GameObject's* position. In this case, where a *Collider* has no companion *RigidBody*, the *Collider* acts as a trigger to a collision and has no interaction on the *GameObject*.

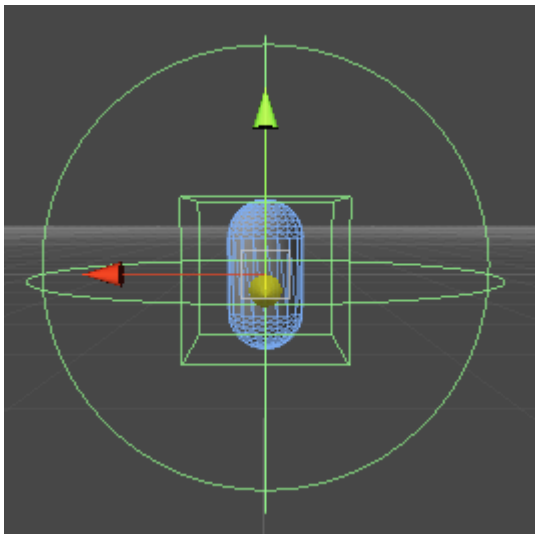


Figure 52: Trigger and Physics Collider Combined

Collision

The *Collision* object is the primary data object used to track a collision between two *Colliders*.

- **Data**
  - Collider : *Collider*
    - The *Collider* that the current collision occurred with
  - Transform : *Transformation*
    - The world *Transformation* of the *GameObject* that the *Collider* collided with.
  - GameObject : *GameObject*
    - The *GameObject* that the *Collider* collided with.

Collider

A base class of all *Colliders*. *Colliders* are the means of which the system is able to manage and track the collisions between *GameObjects*.

Note: In order for two or more objects to collide, they both must have some form of a Collider(box, sphere, etc.) attached to them.

- **Data**
  - IsTrigger : *Boolean*
    - Whether or not the *Collider* is set to act as a trigger.
  - AttachedRigidBody : *RigidBody*
    - The *RigidBody* that the *Collider* is partnered with.
- **Operations**
  - RegisterCallback(callback : *ColliderCallback*) : *void*
    - Registers a callback that will be called when a collision is detected.
  - OnCollisionEnter(collision : *Collision*) : *void*
    - Called when a the collision between two *Colliders* has just started to happen and the *Colliders* have "Entered" each other's bounding volume.
  - OnCollisionExit(collision : *Collision*) : *void*

        - Call when a collision between two *Colliders* has just ended and the *Colliders* have "Exited" each other's bounding volume. This can only be called if a previous *OnCollisionEnter* has been called.
    - OnCollisionStay(collision : *Collision*) : *void*
        - Called when a collision between two *Colliders* is still occurring and the *Colliders* are occupying each other's bounding volume. This can only be called if a previous *OnCollisionEnter* has been called.

BoxCollider

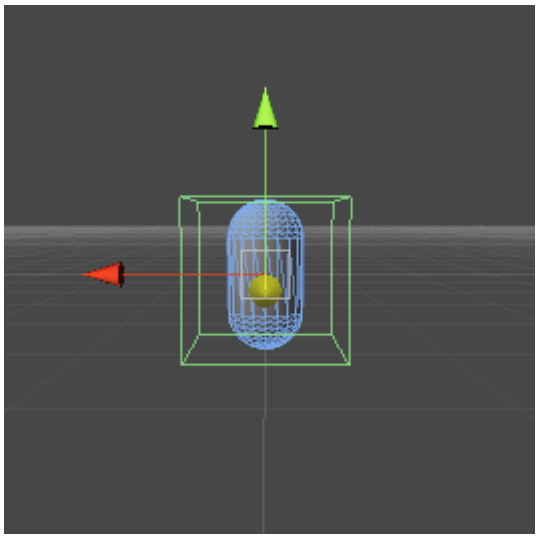


Figure 53: Box Collider Diagram

The *BoxCollider* is a simple axis aligned bounding box collision component.

- **Data**
    - Center : *Vector3*
        - The center of the box, measured in the *GameObject's* local space.
    - Size : *Vector3*
        - The size of the box, measured in the *GameObject's* local space.

SphereCollider

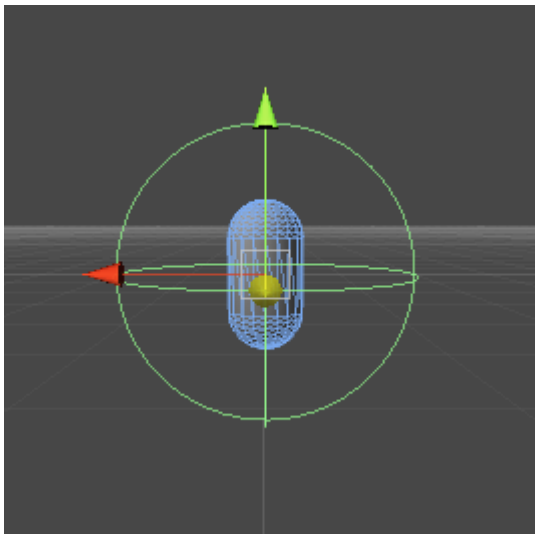


Figure 54: Sphere Collider Diagram

The Sphere Collider is a simple bounding sphere collision component.

- **Data**
  - Center : *Vector3*
    - The center of the sphere, measured in the *GameObject's* local space.
  - Radius : *float*
    - The radius of the sphere, measure in the *GameObject's* local space

CollisionExtension

The *CollisionExtension* is the primary task that splits up and executes the collision tests for all of the registers *Colliders*. The extension uses the calls the *Scene* object to create subsets of *Colliders* that have no dependences, once this is completed each subset can be queued up as a task and run independently.

- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Spawns the a multitude of sub-tasks to check the collisions of all the *Colliders*.

Physics Subsystem

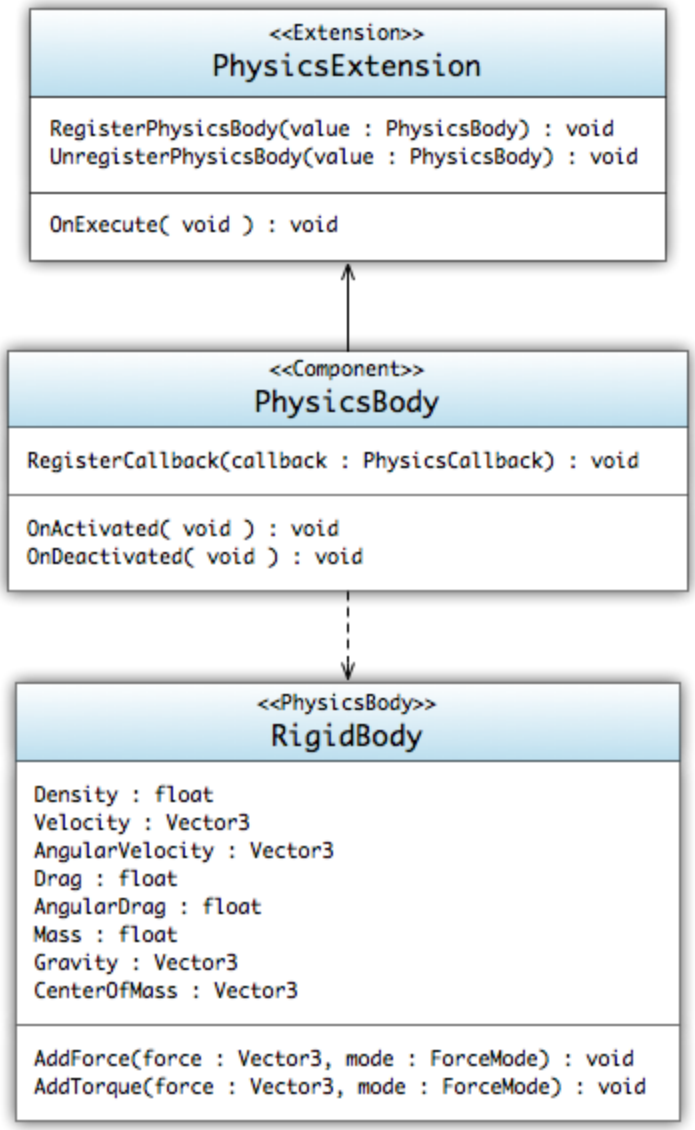


Figure 55: Physics UML Diagram

The physics system is a basic Newtonian physics simulator. In conjunction with the collision system, the physics system is able to manage all of the interactions between multiple collidable objects. During the game, any objects that have an attached *PhysicsBody* will be simulated using the methods defined by the derived classes. In the case of Singularity, which only has the *RigidBody* simulator, the only physics being simulated are simple rigid body physics. At the moment the need for an outside physics engine does not seem necessary, however, if time or design dictate, then the current design will be replaced with the use of a third-party physics engine.

PhysicsBody

Base class for all physics components.

- **Operations**
  - RegisterCallback(callback : *PhysicsCallback*) : *void*

        - Registers a method callback that will be called when the *PhysicsBody* is executed

RigidBody

Controls the GameObjects position through basic physics simulations.

- **Data**
  - Velocity : *Vector3*
    - The velocity vector of the *RigidBody.*
  - AngularVelocity : *Vector3*
    - The angular velocity vector of the *RigidBody.*
  - Drag : *float*
    - The drag associated with the *RigidBody.*
  - AngularDrag : *float*
    - The angular drag associated with the *RigidBody.*
  - Mass : *float*
    - The mass of the *RigidBody.*
  - Gravity : *Vector3*
    - The gravity vector of the *RigidBody.* If no gravity is wanted then set to Vector3(0,0,0).
  - CenterOfMass : *Vector3*
    - The center of mass relative to the transform's origin.

- **Operations**
  - AddForce(force : *Vector3*, mode : *ForceMode*) : *void*
    - Adds a force to the *RigidBody.*
  - AddTorque(force: *Vector3*, mode : *ForceMode*) : *void*
    - Adds a torque force to the *RigidBody.*
  - AddForceAtPosition(force : *Vector3*, position : *Vector3*, mode : *ForceMode*) : *void*
    - Applies a force at the specified position; the force will invoke a torque force as well on the object.
  - AddExplosionForce(force : *float*, position : *Vector3*, radius : *float*, mode : *ForceMode*) : *void*
    - Applies a force to the *RigidBody* that simulates the explosive force.

PhysicsExtension

The *PhysicsExtension* is the primary task used to simulate the physics on *RigidBody* components. With the help of the *Scene* object this extension spawns a number of subtasks to simulate the physics.

- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Spawns the a number of sub-tasks to simulate the physics.

Animation Subsystem

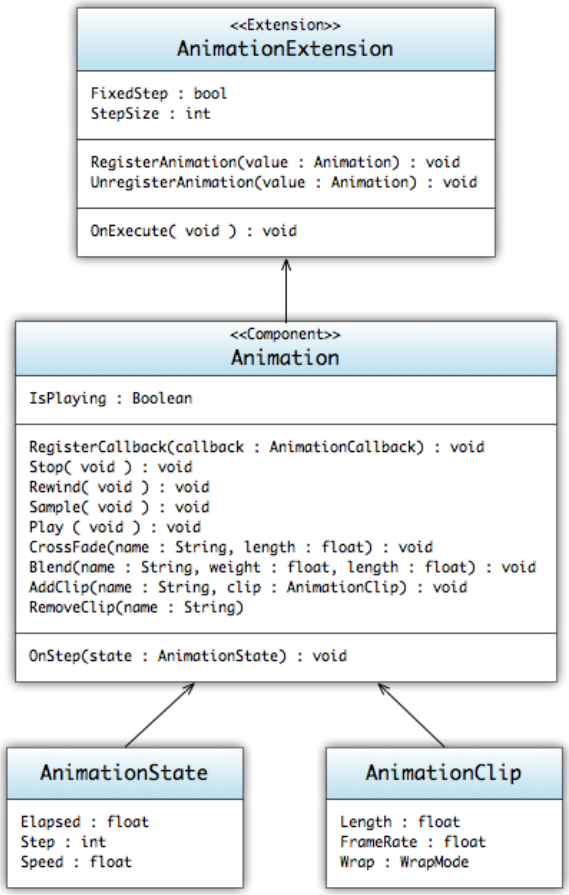


Figure 56: Animation UML Diagram

The animation subsystem is a composition of classes that utilize the built in animation formats, weighted keyframes, of the FBX file format. Although animation is primarily controlled via the loaded in approach, via models, the *Animation* component allows for uses to attach a callback to either calculate pre-processing or to actually animate the *GameObject*. The core object, the *AnimationClip*, is an Object used to store and playback the animation sequence defined by the model format. Due to its nature, the Component can be extended to add inverse-kinematics or any other animation systems if the game development requires it.

Note: Since most animation sequences are directly tied to the render system, the AnimationExtension registers itself as a parent dependency of the RenderExtension.

AnimationClip

Stores the keyframes for the animation clip. The keyframes are loaded from the FBX model definition.

- **Data**
  - Length : *float*
    - Length of the animation clip in seconds.
  - FrameRate : *float*
    - Frame rate at which keyframes are sampled.

- Wrap : *WrapMode*
  - Sets the default wrap mode used in the animation state.

Animation

The *Animation* component is used to play back *AnimationClips*. The animation system is weight based and supports animation blending, additive animations, animation mixing, and full control over all aspects of the animation playback.

- **Data**
  - Clip : *AnimationClip*
    - The default animation clip
  - IsPlaying : *Boolean*
    - Whether or not an *AnimationClip* playing
- **Operations**
  - RegisterCallback(callback : *AnimationCallback*) : *void*
    - Registers a callback to be called when the *Animation* component is executed
  - Stop(*void*) : *void*
    - Stops all playing animations that were started.
  - Rewind(name : *String*) : *void*
    - Rewinds the specified *AnimationClip*
  - Sample(*void*) : *void*
    - Samples the animation at the current state.
  - Play(name : *String*) : *void*
    - Plays the specified *Animation*, if no name is provided then the default animation is started.
  - CrossFade(name : *String*, length : *float*) : void
    - Fades the current animations with the specified *Animation* over a period of time.
  - Blend(name : *String*, weight : *float*, length : *float*) : *void*
    - Blends the specified *Animation* into the currently running animations.
  - AddClip(name : *String*, clip : *AnimationClip*) : *void*
    - Adds an *AnimationClip* to the *Animation.*
  - RemoveClip(name : *String*) : *void*
    - Removes an *AnimationClip* from the *Animation.*

AnimationExtension

The *AnimationExtension* is the primary task used to step the animations through the animation clips. Due to the independent nature of animations, the animation processing is instantly split into as many as ten separate running tasks to execute the animation sequences in parallel.

- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Spawns the *n* number of animation tasks.

Networking Subsystem

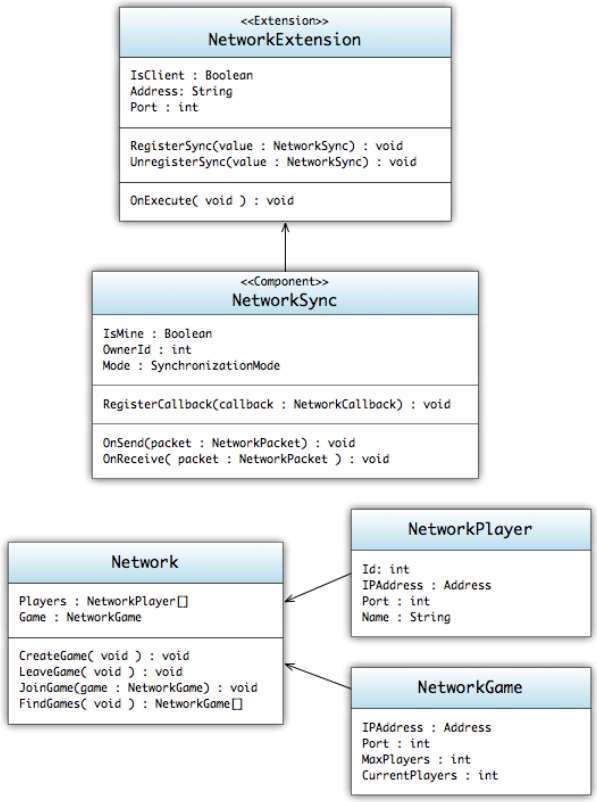


Figure 57: Networking UML Diagram

The networking system runs on multiple multicast networks to distribute and maintain synchronization among all of the connected clients. When the engine starts up it attaches to the "game" network. This network used to publish open games and other higher level commands between clients. The information provided on this network is what is used in the game lobby so that users can choose which games they would like to join. Once joined, or if the player creates a game, a secondary multicast network is created to manage only information pertinent to that instance of a game. The *NetworkSync* works on this game dependent network and utilizes the small multicast network to synchronize objects between all of the clients. If a player leaves, the update is register back on the "game" network and updated; if the last player leaves, then the game is registered as destroyed and the network shutdown and unregistered from the list of available games.

Network

The *Network* class is an interface to the networking communication system. The class provides information and methods to allow the player to join, leave, and create games.

- **Data**
  - Players : *NetworkPlayer[]*
    - The current players in currently running game; if no game has been joined, then the list is empty
  - Game : *NetworkGame*
    - The currently running game; if no game has been joined, then return NULL.

- **Operations**
  - CreateGame(*void*) : *void*
    - Creates a new game and publishes it to the game network.
  - LeaveGame(*void*) : *void*
    - Leaves the joined game.
  - JoinGame(game : *NetworkGame*) : *void*
    - Joins a game; updating the number of players and initializing the in-game network.
  - FindGames(void) : *NetworkGame[]*
    - Finds currently open games on the game network.

NetworkPlayer

The *NetworkPlayer* encapsulates the information needed to identify and track a networked player.

- **Data**
  - IPAddress : *Address*
    - The IP address of the networked player.
  - Port : *int*
    - The port of the networked player.
  - Name : *String*
    - Friendly name or username of the player.

NetworkGame

The *NetworkGame* encapsulates the information needed to identify and track a running networked game.

- **Data**
  - IPAddress : *Address*
    - The IP address of the network game.
  - Port : *int*
    - The port of the networked game.
  - MaxPlayers : *int*
    - The max number of players allowed in the game.
  - CurrentPlayers : *int*
    - The current number of players joined in the game

NetworkSync

The *NetworkSync* is a component that when attached to a *GameObject* will sync the information from other components and transform information to other clients of the game. There are two modes of synchronization, Reliable(TCP) and Unreliable(UDP). Depending on what mode is selected , it will determine the protocol that is used to transfer the information between clients.

- **Data**
  - IsMine : *Boolean*
    - Whether the object being synchronized is the responsibility of the current client.
  - OwnerId : *int*
    - The ID of the *NetworkPlayer* who has responsibility of the object.
  - Mode : *SynchronizationMode*

        - This sets which protocol(UDP/TCP) the network uses to synchronize the information between clients.
- **Operations**
    - RegisterCallback(callback : *NetworkCallback*) : *void*
        - Registers a callback that is called whenever a synchronization happens between the client and host.
    - OnSend(packet : *NetworkPacket*) : *void*
        - Called when a synchronization packet is sent out.
    - OnReceive(packet : *NetworkPacket*) : *void*
        - Called when a synchronization packet is received.

NetworkExtension

The *NetworkExtension* manages the connection between both the "Game" network and the game instance networks. When run, the extension will spawn off packet builder tasks. Each packet builder task will collect the synchronization information needed to be sent to other clients. Once the packet data is collected, the system will bundle that up and send it out to the other game clients.

- **Operations**
    - OnExecute(*void*) : *void*
        - Inherited from *Task*; Collects the synchronization data and then packages it up and sends it out to game clients.

Lua Scripting

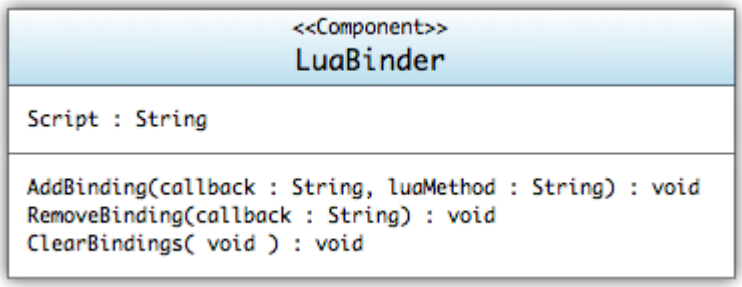


Figure 58: Lua UML Diagram

The Lua script interpreter is one of the main ways for designers/developers to modify object behaviors. Instead of building off of the *Extension/Component* architecture of most subsystems, the lua subsystem uses a single *Component*, the *LuaBinder*. The main purpose of the *LuaBinder* is to manage the bindings between *Component* callbacks and the Lua script methods.

LuaBinder

The *LuaBinder* is a *Component* used to bind Lua snippets to the *GameObject* and *Component* Method callbacks. Execution of the Lua script is run immediately and is given access to the objects scope. Management of the *LuaState* is held through global variables and management across all *LuaBinders*.

- **Data**
    - Script : *String*
        - Lua Script file to reference

- **Operations**
  - AddBinding(callback : *String*, luaMethod : *String*) : *void*
    - Adds the callback binding to the *LuaBinder*; when the callback is called, the method will be redirected to the lua scripted method.
  - RemoveBindings(callback : *String*) : *void*
    - Removes the callback binding
  - ClearBindings(*void*) : *void*
    - Clears all of the callback bindings.

Particles Subsystem

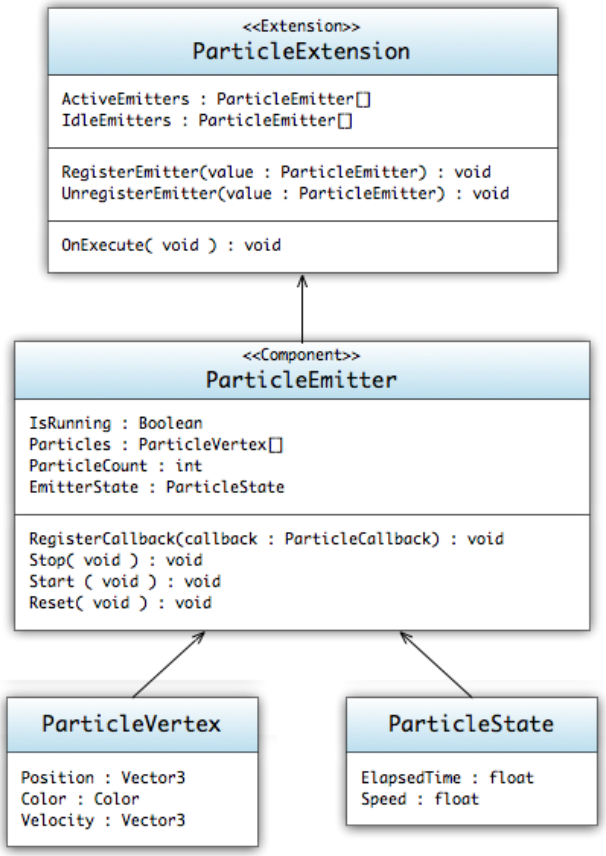


Figure 59: Particle UML Diagram

The particles subsystem is a composition of classes that will render particle effects to the screen. The subsystem will be managed by a *ParticleExtension* which handles all particle emitters created, running, and idling through the game. The *ParticleExtension* will have a series of *ParticleSubTasks* that will divide up the tasks for the *TastExtension. ParticleEmitters* will be created and attached to *GameObject*s and from there will handle the rendering of the particles. The emitters will store *ParticleVertexs* and *ParticleState*s to assist in keeping track of the progress of the particle rendering.

ParticleState

The *ParticleState* will be responsible for keeping track of the time along the emitter's lifespan. The timer's data will translate into the *ParticleVertex* itself via particle age.

- **Data**

- ElapsedTime : *float*
  - Amount of time that has passed since the emitter has started.
- Speed : *float*
  - The rate at which the timer ticks

ParticleVertex

The *ParticleVertex* will hold all the possible data that a particle will need to render. This data is passed into the *VertexBuffer* and to the *GraphicsDevice*. Shaders will take the data in from the vertices and render them out to the screen.

- **Data**
  - Position : *Vector3*
    - The starting position of the vector
  - Color : *Color*
    - Color of the vertex
  - Velocity : *Vector3*
    - The velocity that alters the position of vertex over time.

ParticleEmitter

The *ParticleEmitter* is used to render particle effects. The emitter will keep track of a vertex buffer which will house all of the *ParticleVertexes* to be rendered. Emitters attach themselves to *GameObjects* as components so they can be recognized and rendered by the renderer. Emitters will need to keep track of their overall lifetime and translate that information to the particles.

- **Data**
  - IsRunning : *Boolean*
    - Whether or not the *ParticleEmitter* is currently running.
  - Particles : *ParticleVertex*[]
    - Array of particle vertices that will be passed into the vertex buffer
  - ParticleCount : *int*
    - The max number of particles the emitter can hold
  - EmitterState : *ParticleState*
    - Timer object that keeps track of the emitter's lifespan
- **Operations**
  - RegisterCallback(callback : *ParticleCallback*) : *void*
    - Registers a callback event whenever the particle emitter component needs to be rendered.
  - Stop(*void*) : *void*
    - Stops the *ParticleEmitter* from moving to the next display state.
  - Start(*void*) : *void*
    - Starts the *ParticleEmitter*.
  - Reset(*void*) : *void*
    - Resets the *ParticleEmitter* to its initial state.

ParticleExtension

The *ParticleExtension* is the workhorse of the particle subsystem. It is required to manage all active and idling emitters. When a call is made for a *ParticleEmitter* to be created, it is the extension's job to create this object. It is also the emitter's responsibility to kill any active emitters when their lifespan has

passed. When an emitter's lifespan reaches its max, it is taken out of the active queue and placed into an idle queue to be used later. This helps with reducing the need to create new emitters as a game progresses.

- **Data**
  - ActiveEmitters : *ParticleEmitter[]*
    - Full list of emitters that are currently active in the system
  - IdleEmitters : *ParticleEmitter[]*
    - List of emitters that have gone through the active system and are currently waiting to be used again
- **Operations**
  - OnExecute(*void*) : *void*
    - Inherited from *Task*; Executes and spawns particle sub-tasks.

Audio Subsystem

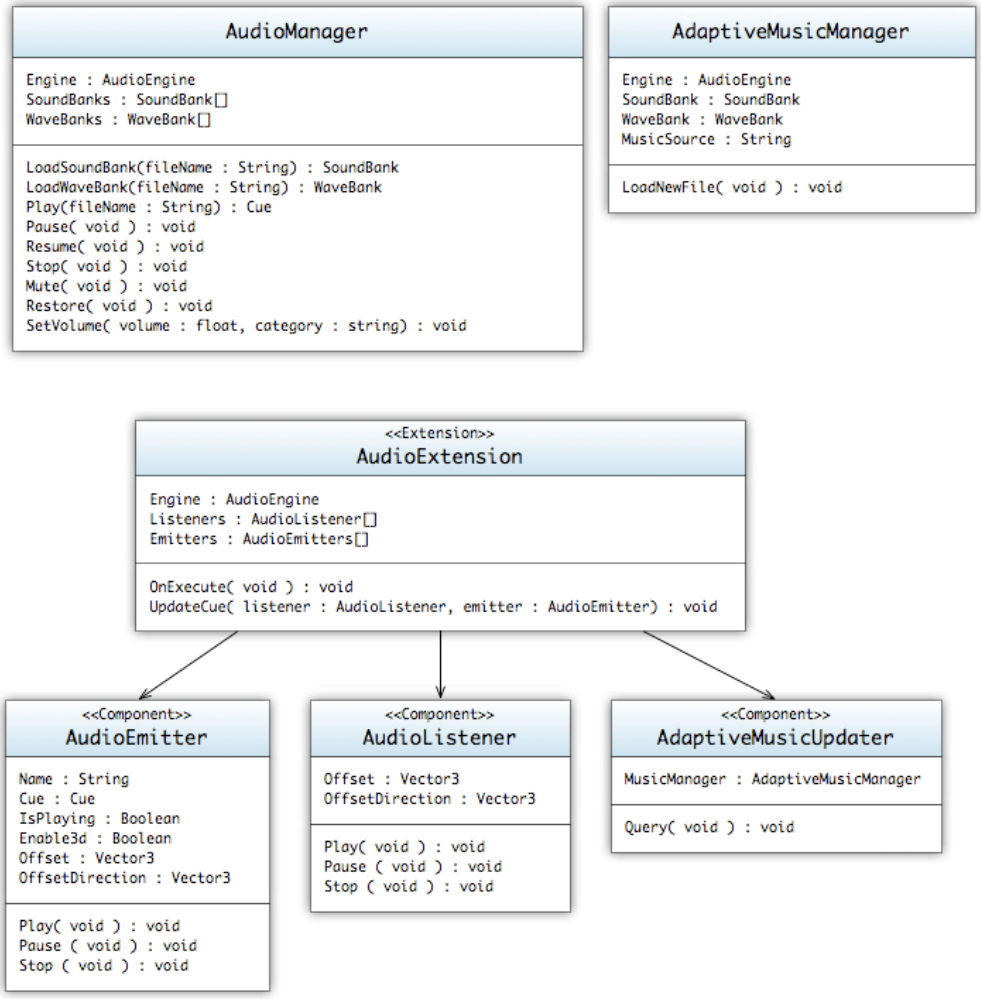


Figure 60: Audio UML Diagram

The audio subsystem allows access to the audio device and allows audio assets to be played. The audio subsystem is hardware-agnostic and utilizes XACT as a wrapper to XAudio2. The *AudioManager* class will be the primary interface used to play a sound. The *AudioManager* will contain a pointer to the XACT audio engine. Assets will be managed with external software and loaded into the system using the Soundbanks and Wavebanks, which wrap XACT's implementations. The system is designed to abstract away the complexities of XACT and allow other programmers to interface simply and easily with audio by playing sound through emitters. Finally, a cue will be used to manage each instance of a sound. Specialized *AudioEmitter* and *AudioListener* components will be placed onto specific *GameObjects* in order to update 3d positional information.

Attached to the audio subsystem is a unique music management system for adaptive audio. The *AdaptiveMusicManager* will require up-to-date information, which will be provided by the *AdaptiveMusicUpdater.* Developers need only interact with the *AdaptiveMusicManager* directly if another set of music should be changed, e.g. if a new level is being loaded.

AudioManager

The *AudioManager* is the main point of access for anything related to sound. It will maintain a collection of *SoundBanks* and *WaveBanks*, as well as a pointer to the *AudioEngine*. This class allows access to playback features such as pause and volume controls. The *AudioManager* can also directly play an audio file without associating it with a cue.

- **Data**
  - Engine : *AudioEngine*
    - The *AudioEngine* used by the system.
  - SoundBanks : *SoundBank[]*
    - The list of SoundBanks in the system.
  - WaveBanks : *WaveBank[]*
    - The list of WaveBanks in the system.
- **Operations**
  - LoadSoundBank(fileName : *String*) : *SoundBank*
    - Loads a SoundBank into the system. Expects a .xsb extension on the filename.
  - LoadWaveBank(fileName : *String*) : *WaveBank*
    - Loads a WaveBank into the system. Expects a .xwb extension on the filename.
  - Play(fileName : *String*) : *Cue*
    - Prepares and plays a particular file. Returns a cue associated with that instance.
  - Pause (*void*) : *void*
    - Pauses or unpauses all playback on the audio engine. This affects all cues.
  - Stop (*void*) : *void*
    - Stops all cues currently being played in the audio engine. They cannot be restarted without being re-prepared.
  - Mute(*void*) : *void*
    - Mutes the audio engine without pausing it. This affects all cues.
  - Restore(*void*) : *void*
    - Unmutes the audio engine without pausing it. This affects all cues.
  - SetVolume(volume : *float,* category : *String*) : *void*
    - Sets the volume for an XACT category defined in the asset management tool.

AudioEmitter

The *AudioEmitter* is a *GameObject* component that contains a *Cue.* The *AudioEmitter* will update the positional data in the *Cue* and will identify when to fire off a cue. Each *AudioEmitter* represents a single instance of a single cue.

- **Data**
  - Name : *String*
    - A name for the cue being emitted.
  - Cue : *Cue*
    - The *Cue* that emits from this *AudioEmitter.*
  - IsPlaying : *boolean*
    - Whether or not the cue is currently active.
  - Enable3d : *boolean*
    - Whether or not the cue is positioned in 3d.
  - Offset : *Vector3*
    - A positional offset of the component from the *GameObject*.

    - OffsetDirection : *Vector3*
        - The direction to offset.
- **Operations**
    - Play(*void*) : *void*
        - Begins to play the cue. Unpauses if paused.
    - Pause(*void*) : *void*
        - Pauses or unpauses the cue.
    - Stop (*void*) : *void*
        - Stops playing the cue and reprepares it.

AudioListener

The *AudioListener* is a *GameObject* component that attaches to a *GameObject* and utilizes its position for 3D audio calculations.

- **Data**
    - Offset : *Vector3*
        - A positional offset of the component from the *GameObject*.
    - OffsetDirection : *Vector3*
        - The direction to offset.

AdaptiveMusicUpdater

The *AdaptiveMusicUpdater* is a *GameObject* component that attaches to a *GameObject* and can update the *AdaptiveMusicManager* with game state information.

- **Data**
    - MusicManager : *AdaptiveMusicManager*
        - The adaptive music manager this component will update.
- **Operations**
    - Query(*void*) : *void*
        - Queries the *GameObject* it is attached to and updates the *AdaptiveMusicManager*.

AdaptiveMusicManager

The *AdaptiveMusicManager* is a special class designed to control the background music of the game. While linear audio is played through the traditional *AudioManager*, the *AdaptiveMusicManager* takes a collection of "slices" of audio and sews them together. The *AdaptiveMusicExtension* will update the *AdaptiveMusicManager* when necessary.

- **Data**
    - SoundBank : *SoundBank*
        - The soundbank associated with the adaptive music.
    - WaveBank : *WaveBank*
        - The wavebank associated with the adaptive music.
    - Engine : *AudioEngine*
        - The audio engine associated with the adaptive music.
    - MusicSource : *String*
        - The path to an XML file in which the music information is located.
- **Operations**
    - LoadNewFile(*void*) : void

        - Reads in adaptive audio information based upon the source XML file in question.

AudioExtension

The *AudioExtension* updates all of the audio information in the game world. It is comprised of three types of subtasks: one to update all of the positional audio information, one to update the audio engine once per frame, and one to query the game world and update the *AdaptiveAudioManager*. Ten subtasks will be spawned to manage the *AudioListener*/*AudioEmitter* calculations, and a single subtask will be provided to each of the other two types.

- **Data**
    - Engine : *AudioEngine*
        - The audio engine. Used to update the 3d cue information.
    - Listeners : *AudioListener[]*
        - The components that represent audio listeners.
    - Emitters : *AudioEmitter[]*
        - The components that represent audio emitters.
- **Operations**
    - OnExecute(*void*) : *void*
        - Inherited from *Task*. Spawns a collections of subtasks to update all of the information required for the audio engine, 3d position audio, and the adaptive audio system.
    - UpdateCue(listener *: AudioListener,* emitter : *AudioEmitter*) : *void*
        - Updates the DSP information for the cue in the audio engine. Calculates and applies all positional data.

GUI Subsystem

The GUI subsystem is used to create any type of 2D interface for the user. Examples include title screen, match lobbies, and HUDs. Each screen is attached to a camera and enabled or disabled depending on what parts are allowed to render to the screen. Like all of the other components, this system has a background extension that changes the devices render state to allow 2D rendering with alpha blending of the background. Due to the blending, GUIs are usually placed on the last layer of the rendering sequence in order to prevent transparency artifacts.

GUIScreen

The *GUIScreen* is the main render point for generating a GUI for the camera. All of the controls are added to this screen and positioned. If a *GUIScreen* is enabled it will be drawn as a 2D orthogonal plane overtop all of the rest of the scene.

- **Data**
    - Camera : *Camera*
        - The *Camera* the *GUIScreen* is attached to for rendering
    - Controls : *Control[]*
        - The list of *Controls* added to the *GUIScreen*.
- **Operations**
    - AddControl(control : *Control*) : *void*
        - Adds a control to the *GUIScreen* and causes the control to render when the camera screen is rendered.
    - RemoveControl(control : *Control*) : *void*

        - Removes a control from the list of added controls; no longer rendering it on the camera screen.
    - Clear(*void*) : *void*
        - Clears all of the controls from the *GUIScreen.*

NinePatch

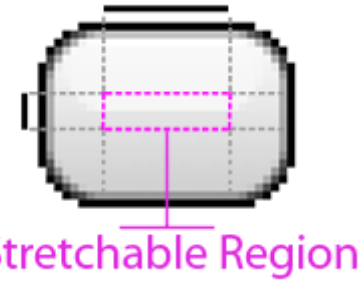


Figure 61: NinePatch Diagram

The GUI system supports a stretchable bitmap image, called a 9-Patch graphic. This is a PNG image in which you define stretchable sections that the GUI system will resize to fit the object at display time to accommodate variable sized sections, such as text strings.

- **Data**
    - StretchableRegion : *Rectangle*
        - Defines the left, top, right, and bottom of the stretchable region of the texture.
    - Size : *Size*
        - The size of the texture.

Font

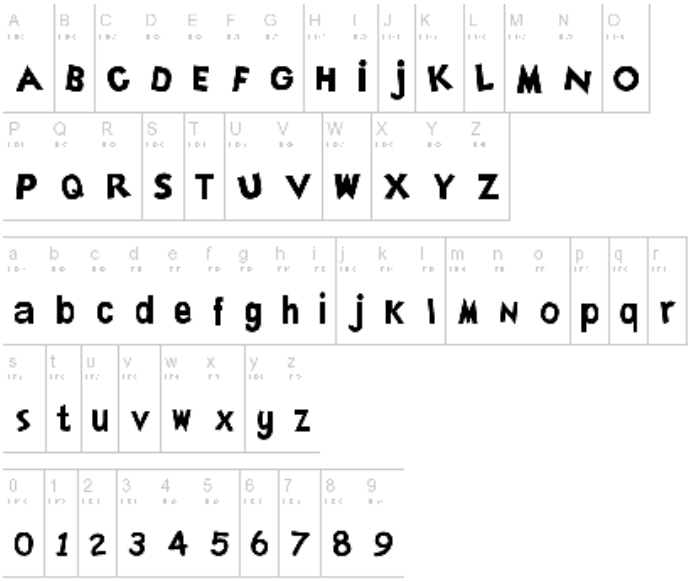


Figure 62: Font Map

The *Font* object takes font map textures that defines the character layout and sizing of a specific font. All controls take a font object when displaying any form of text.

- **Data**
  - FontMap : *Texture*
    - The font map *Texture* of the *Font.*

Control

Represents the base class for user interface (UI) elements that use *Texture* and *Fonts* to define their appearance.

- **Data**
  - Font : *Font*
    - The font object to use when printing out the specified text.
  - Text: *String*
    - The text to display; the means of display changes depending on the type of *Control.*

Panel

Figure 63: Panel Control Diagram

A *Panel* is a control that contains other controls. You can use a *Panel* to group collections of controls such as a group of Labels or Button controls. The *Panel* control is displayed by default without any texture or output rendering. You can provide a *NinePatch* texture to enable border and background rendering.

- **Data**
  - Texture : *NinePatch*
    - The *NinePatch* texture to apply to the background of the *Panel.*
  - Padding: *Rectangle*
    - Defines the amount of spacing around the border that is given left empty when drawing inner controls.
  - Controls : *Control[]*
    - The list of *Controls* added to the *Panel*.
- **Operations**
  - AddControl(control : *Control*) : *void*
    - Adds a control to the *GUIScreen* and causes the control to render when the camera screen is rendered.
  - RemoveControl(control : *Control*) : *void*

        - Removes a control from the list of added controls; no longer rendering it on the camera screen.
    - Clear(*void*) : *void*
        - Clears all of the controls from the *GUIScreen*

Label

Figure 64: Label Control Diagram

Represents the text label for a control, the control has no interactions only display properties.

Button

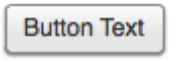


Figure 65: Button Control Diagram

A *Button* is a control that allows user interaction through the mouse/keyboard interface. The *Button* control is displayed by default with a basic texture. You can provide a *NinePatch* texture to change the border and background rendering.

The following example shows three buttons that respond to clicks in three different ways.

- **Hover**: the first button changes colors when the user hovers with the mouse over the button.
- **Press**: the second button requires that the mouse be pressed while the mouse pointer is over the button.
- **Release**: the third does not reset the background color of the buttons until the mouse is pressed and released on the button.

- **Data**
    - Texture : *NinePatch*
        - The *NinePatch* texture to apply to the background of the *Panel.*
    - Padding: *Rectangle*
        - Defines the amount of spacing around the border that is left empty when drawing the text.
    - Callback : *ButtonCallback*
        - Sets the function callback that is called when a user presses the button.

Textbox

Figure 66: Textbox Control Diagram

A *Textbox* is a control that allows user interaction through the keyboard interface. The *Textbox* control is displayed by default with a basic texture. You can provide a *NinePatch* texture to change the border and background rendering.

- **Data**
    - Texture : *NinePatch*
        - The *NinePatch* texture to apply to the background of the *Panel.*
    - Padding: *Rectangle*

        - Defines the amount of spacing around the border that is given left empty when drawing the text.
    - Callback : *TextboxCallback*
        - Sets the function callback that is called when a user presses a key within in the *Textbox*.

Render2DExtension

The *Render2DExtension* is the primary task used to render GUI controls to the camera screen. This extension spawns a multitude of subtasks to render out the *Controls* using the *BufferedDeviceAdapter*; once the tasks have completed, the *BufferedDeviceAdapters* are combined and played back using the *HardwareDeviceAdapter*.

- **Operations**
    - OnExecute(*void*) : *void*
        - Inherited from *Task*; Spawns the a multitude of sub-tasks to render the *Controls* to the camera screen.

Content Subsystem

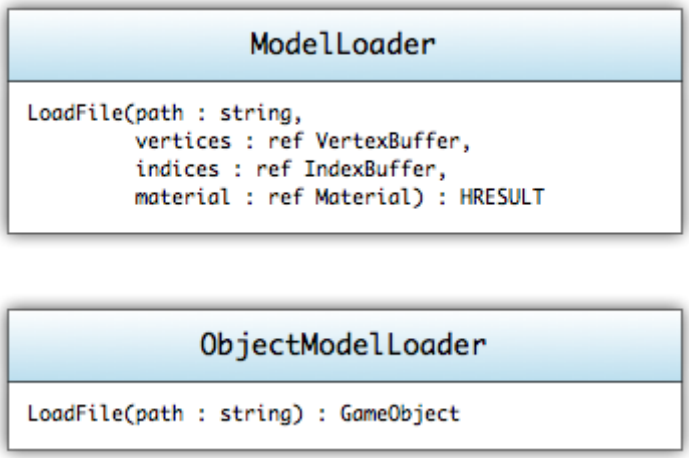


Figure 67: Content Loader UML Diagram

The component subsystem is a set of static tools used to load models and game objects into the game. These tools make use of common formats(FBX, OBJ, etc.) in order to load the vertices, indices, materials and textures needed to draw the objects. The extended *ObjectModelLoader* uses a system of attributes embedded within the FBX to enable the loader to create the *GameObject* and attach the needed components and properties.

ModelLoader

Static content tool used to load in model files and return back the imported graphics components. If the system is unable to load in aspects of the model, certain buffer types will be returned as NULL.

- **Operations**
    - LoadFile(path : *String*, vertices : *ref VertexBuffer,* indices : *ref IndexBuffer*, material : *ref Material) : HRESULT*
        - Used to load the file and return the imported *VertexBuffer*, *IndexBuffer*, and *Material.*

ObjectModelLoader

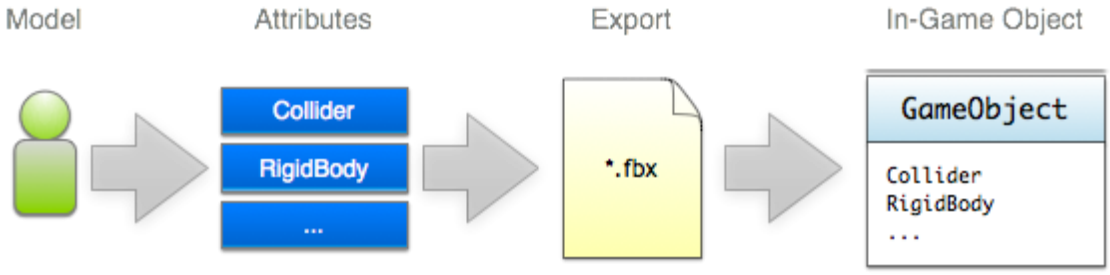


Figure 68: Object Import Process

The process for loading in an object into the game is designed to take advantage of the FBX's ability to append attributes to objects. Once a model is completed, different attributes can be added to the model in order to control the behaviors when loaded into the engine. Through the use of the Maya/Singularity Component plugin, we can control different component behaviors and design through the Maya designer interface. Once exported, the attributes are embedded and finally read into the system by the *ObjectModelLoader*.

Static content tool used to load in object definition files and return back the *GameObject* with all of the detail *Component* elements added and filled in with the attribute values. If the system is unable to load in aspects of the object, the function will return NULL.

- **Operations**
  - LoadFile(path : *String) : GameObject*
    - Used to load the file and return the imported *GameObject* with all of the *Component* objects detailed by the attribute system.

Game

Components

These components are specific to the Trigger Happy game and are not initially part of the Singularity Engine's *Component* set.

ResourceManager

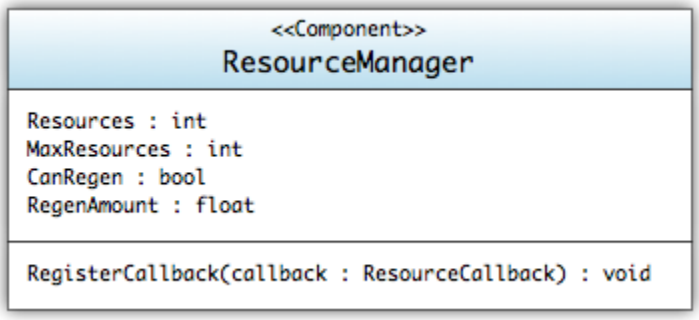


Figure 69: Resource Manager UML Diagram

The *ResourceManager* is a simple component used to manage a form of resource. Examples would include health, ammo, energy, etc.

- **Data**
  - Resources : *int*
    - The number of resources currently available.
  - MaxResources : *int*
    - The max number of resources that can be made available.
  - CanRegen : *Boolean*
    - Can the resource regenerate over time.
  - RegenAmount : *float*
    - The amount of resource added to *Resources* every second.
- **Operations**
  - RegisterCallback(callback : *ResourceCallback*) : *void*
    - Registers a callback that is called whenever the resource value changes.

GameManager

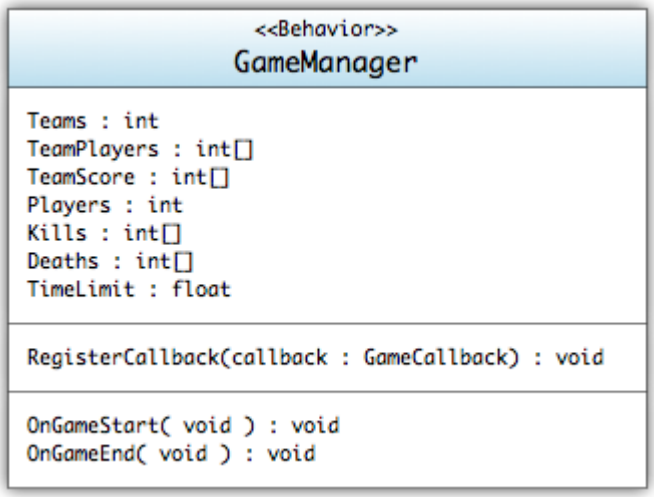


Figure 70: GameManager UML Diagram

The *GameManager* is a component added at the *Level GameObject* so that information about the type of game and the statistics of the current state can be saved and watched for win/lose conditions. Since the game will have two main types(Assault and King of the Hill) the behaviors of the *GameManager* logic will have to be augmented. In concert with a *LuaBinder* the *GameManager's* callback will be bound to lua scripts that define the conditions for a win. This method of augmenting the runtime will allow the developers to adjust the game parameters at runtime to fine tune the play style and speed.

- **Data**
  - Teams : *int*
    - The number of teams currently participating in the game.
  - TeamPlayers : *int[]*
    - The team that each player is assigned it.
  - TeamScore : *int[]*
    - Keeps track of each teams score.
  - Players : *int*
    - The number of players currently participating in the game.

- Kills : *int[]*
  - The number of kills for each player
- Deaths : *int[]*
  - The number of deaths for each player
- TimeLimit : *float*
  - The max amount of time allotted to the game.

- **Operations**
  - RegisterCallback(callback : *GameCallback*) : *void*
    - Registers a callback that is called each update/render sequence.
  - OnGameStart(*void*) : *void*
    - Called when the game is initialized and started.
  - OnGameEnd(*void*) : *void*
    - Called when the game is finished.

FirstPersonCamera

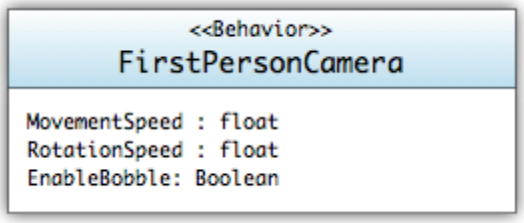


Figure 71: FirstPersonCamera UML Diagram

The *FirstPersonCamera* is a child of the *Camera* class. In addition to the *Camera's* render functionality, the *FirstPersonCamera* adds input based movement. Movement uses the standard keyboard(WASD) and mouse combination popular to most first-person shooters. Attaching the component to a GameObject causes the *FirstPersonCamera* sets itself as the primary camera and all render sequences use the *FirstPersonCamera* as its render output.

- **Data**
  - MovementSpeed : *float*
    - The speed that the *Camera* will move when the player presses any of the directional keys.
  - RotationSpeed : *float*
    - The speed that the *Camera* will rotate when the player moves the mouse.
  - EnableBobble: *Boolean*
    - Whether or not movement will bob the camera up and down.

ModifierEffect

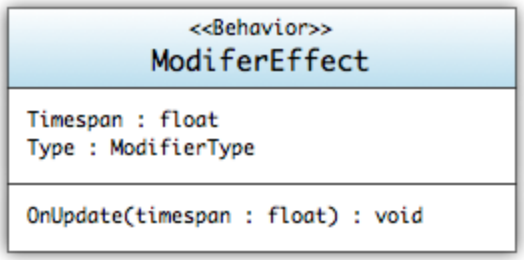


Figure 72: ModifierEffect UML Diagram

The *ModifierEffect* is used to attach effects to the players when they activate a *Modifier*. The *ModifierEffect* has a duration timer that when reached will automatically remove the effect from the player.

- **Data**
  - Timespan : *float*
    - The duration length of the applied effect.
  - Type : *ModifierType*
    - The type of modifier effect to apply.
- **Operations**
  - OnUpdate(timespan : *float*) : *void*
    - Called on each update sequence.

Game Objects

With the component-based architecture all of the usual game objects are replaced by a aggregation of components attached to a *GameObject*. Because of this, the development of the core objects for the game will be less programming and more design. By adding the LuaBinder to the objects, designers and developers can augment the behaviors of the *GameObject*s with nothing more than lua scripts.

Note: The list of game objects will layout the collection of Components that will be attached and their purposes for inclusion into the object. Additional behaviors, via lua scripting, will be described in the LuaBinder section.

Modifier

The *Modifier* game object is one of the primary weapon types used in the game. Modifiers are analogous to "remote mines" in that they are player activated and must be placed throughout the game play. When players walk too close to the modifiers, the owner of the modifier can activate it and have a specific effect applied to the other player. The effect is managed by adding a *ModifierEffect* component to the other player. Once said effect has timed out, the *ModifierEffect* will remove itself from the *GameObject* thereby removing the once present effect on the player.

Figure 73: List of components attached to a modifier

MeshRenderer

The *MeshRenderer* is used to store and render the *Modifier's* model. The *Material* applied will control whatever render effects are to be drawn as decided by the art direction.

BoxCollider

The *BoxCollider* is added to aid in the "throwing" of the *Modifier*. Since the device is not active until it attaches to a surface, the *BoxCollider* is used to 1) prevent the object from falling through the floors, ceilings, and walls and 2) being notified when it has reached a surface so that it can attach at the contact point.

RigidBody

The *RigidBody* is used much like the *BoxCollider* in that its main purpose for adding is during the "throwing" phase of the *Modifiers* lifecycle. The *RigidBody* component will make sure that the object falls in an "arc" depending on the forces applied.

Animation

When a *Modifier* is activated, an quick animation will play to display to the user what has just happened. In order to play through the animation sequence, the *Animation* component is added and populated with the animation sequences.

ParticleEmitter

Some *Modifier* types, when activated, have a secondary animation sequence to show the player what type of effects are being applied. The *ParticleEmitter* will be used to show effects such as the inverse gravity, grow, and shrink. All of these are just another way of showing to the player and those around him what effects are being applied.

NetworkSync

Since Trigger Happy is a networked multiplayer game, object need to be synchronized and managed across multiple clients. In order to accomplish this, the *NetworkSync* is attached and configured to synchronize state and position to all of the other clients connected to the current game.

LuaBinder

The *LuaBinder* is the main means for the developer to add the custom behaviors that a *Modifier* exhibits. Behavior scripts would manage the following...

- What particle effect to apply for which type of modifier
- Whom to apply the *ModifierEffect* to when the *Modifier* is activated.
- Enabling the SphereCollider once the "build" time has passed.

SphereCollider

In order to facilitate the activation check the *SphereCollider* is added. The purpose of this *Collider* is to act as a trigger. When someone enters the bounding volume of the *SphereCollider* no physical actions will be taken, but the callback will be called and the observing class will be notified that someone or something has entered within the activation radius.

AudioEmitter

Each modifier type has a unique sound that it plays for player recognition. In addition, each *Modifier* has a different sound (and therefore emitter) that will be triggered at certain conditions. Because *Modifiers* exist in 3d space, the *AudioEmitter* allows for movement of the source sound for 3d positional audio.

Weapon

Weapons are the core to the game play and as such have a very interactive role with the play of the game. Each weapon has a specific set of attributes, each of which dictates how it fires, what the projectiles look like, and what the damage is. Much like the *Modifiers* the *Weapon* has most of the general components needed to exist in the virtual world. It lacks any physics additions because the *Player* or *Projectile*, not the *Weapon,* will actually interact with the level and objects stored within.

Figure 74: List of components attached to weapons

MeshRenderer

The *MeshRenderer* is used to store and render the *Weapons's* model. The *Material* applied will control whatever render effects are to be drawn as is decided by the art direction.

Animation

When a *Weapon* is fired, a quick animation will play. In order to play through the animation sequence, the *Animation* component is added and populated with the animation sequences.

ParticleEmitter

Some *Weapon* types, when fired have a secondary animation sequence in the form of particle effects. The *ParticleEmitter* is attached to be able to display the *Weapon's* particle effects.

ResourceManager

Since all *Weapons* have some form of cool down and ammo resource, a *ResourceManager* is added to help track and publish the values associated with the ammo. The *ResourceManager* is just a data component and its only purpose is to provide a public front-end to the ammo values.

LuaBinder

The *LuaBinder* is the main means for the developer to add the custom behaviors that a *Weapon* exhibits. Behavior scripts would manage the following...

- Creation and initialization of projectile firing
- Fire rate via the *ResourceManager*
- Type of model/projectile that is valid for the specific *Weapon* type

AudioEmitter

Each Weapon has a unique set of sounds that it plays. Each sound is triggered based off a specific condition. Because Weapons exist in 3d space, the AudioEmitter allows for movement of the source sound for 3d positional audio.

Projectiles

Projectiles are the objects fired from *Weapons.* Each projectile follows basic physics and depending on mass and velocity will behave appropriately. When a *Projectile* collides with another collidable object, the *Projectile* will destroy itself and apply whatever damage is has to the collision space.

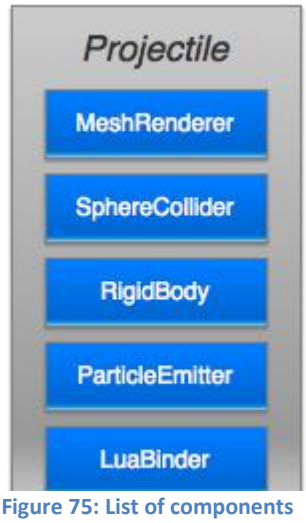


Figure 75: List of components attached to projectiles

MeshRenderer

The *MeshRenderer* is used to store and render the *Projectile's* model. The *Material* applied will control whatever render effects are to be drawn as decided by the art direction.

SphereCollider

The *SphereCollider* is added to allow the projectiles to interact with other objects within the world. At any time when another object collides with the projectile, the projectile will destroy itself and apply the damage modifiers to the collided object.

RigidBody

Since *Projectiles* are physics objects, the *RigidBody* is added to simulate the basic physics that is to be applied over the life of the object.

ParticleEmitter

Most *Projectiles* will have some form of vapor trail. In order to display such effect, the *ParticleEmitter* is attached and initialized with the appropriate effects and parameters.

LuaBinder

The *LuaBinder* is the main means for the developer to add the custom behaviors that a *Projectile* exhibits. Behavior scripts would manage the following...

- Damage the collided players and/or objects
- Destruction of the projectile when collision occurs

AudioEmitter

When a projectile collides with another object, a collision sounds needs to be played in 3d space. The AudioEmitter allows that sound to be played in the 3d environment.

StaticObject

StaticObjects are any object in the world that does not have any form of user control. Objects such as the level's wall, boxes and other environment objects all fit into the category of *StaticObjects*.

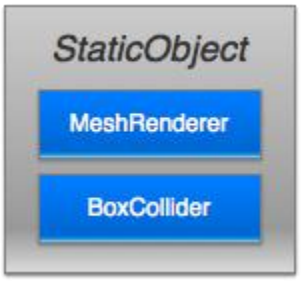


Figure 76: List of components attached to StaticObjects

MeshRenderer

The *MeshRenderer* is used to store and render the *StaticObjects* model. The *Material* applied will control whatever render effects are to be drawn as decided by the art direction.

BoxCollider

The *BoxCollider* is added to interact with other objects such as *Projectiles* and *Players*. Even though no physics effects will be applied to the object, the collision will prevent other objects from occupying the same space as the *StaticObject.*

AudioEmitter

Some *StaticObjects* have associated ambient sounds attached to them. The *AudioEmitter* allows these *StaticObjects* to play this sound in 3d positional space for the player's immersion.

Player

The Player game object is the character interface through which other players are identified or the player interacts with the virtual world. The Player object is one of the most complex and has a multitude of components attached to manage all of the behaviors expected of the play style. Since players need to be represented across all clients, there are two setups for the components that will be attached. The first setup is for the player's actual character, this is the one that they will control and move around the world with. The second one is the networked control object that is being mirrored from another client's system.

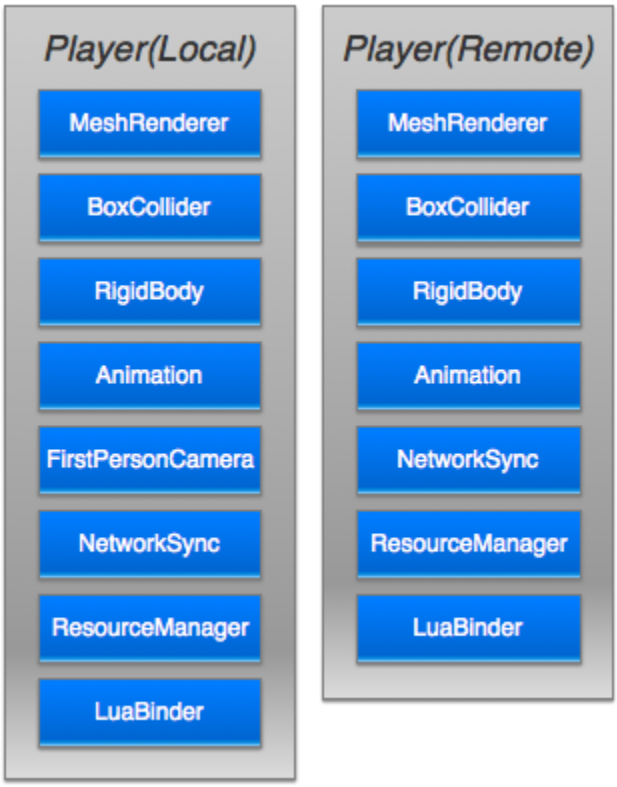


Figure 77: List of components attached to players, both local and remote

MeshRenderer

The *MeshRenderer* is used to store and render the *Player's* model. The *Material* applied will control whatever render effects are to be drawn as decided by the art direction.

BoxCollider

The *BoxCollider* is added so that the *Player* object is able to interact and collide with objects in the virtual world. Without the *Collider*, the player could walk through walls and would fall through the floor. Due to the size and speed of which the game is going to be played, a bounding box should be sufficient for the resolution of collisions without looking awkward.

RigidBody

The *RigidBody* is used in conjunction with the *BoxCollider*. Its main purpose is to allow the basic physics simulations to be applied to the player. By adding physics simulations we are able to allow explosions to knock-back the player or modifier effects to be applied (such as inverse gravity) to the player. Since many of the modifiers are physics based, the need and subsequent use of this component is core to the game play and expected behavior of the players.

Animation

Being that the *Player* is really the main focus of the players, there are a multitude of animations that need to be applied to maintain the game immersiveness. Animations such as walking/running, firing, falling are all stored within the *Animation* component, whenever certain events occur the animation system will blend between animation sequences.

FirstPersonCamera

When the *Player* object is the one being controlled by the player, they will have a *FirstPersonCamera* attached. This allows the player to use the general first-person keyboard(WASD) and mouse controls to move and look around using the *Player*.

NetworkSync

The *NetworkSync* has two functions, when the *Player* is the controlled object the *NetworkSync* is used to send information about the *Player's* position to all of the other clients. All of the other instances of the *Player* objects have the other end of that *NetworkSync*; this means that they will mirror the data received by the game network to the local instance of the object.

ResourceManager

The *ResourceManager* for the *Player* is used to monitor and manage the shield value of the *Player.* Again, like all of the other instances of the *ResourceManager* the main purpose of this object is to provide a public interface to the *Player's* shield values.

LuaBinder

The *LuaBinder* is the main means for the developer to add the custom behaviors that a *Player* exhibits. Behavior scripts would manage the following...

- Handling the switching of weapons
- Changing animation sequences depending on *Player* state

AudioEmitter

Each *Player* has a number of sounds that needs to be played based upon a number of conditions. On a local *Player*, these sounds need not be in 3d, but any remote *Player's* sounds must appear to source from those locations.

AudioListener

The *AudioListener* is the source point for all 3d audio calculations in the 3d environment. The *AudioListener* only needs to be attached to the local *Player*. This component is used to determine distance between sounds and the *Player* as well as how loud and in what direction the sound appears to come from.

Level

The *Level* object is a non-drawable controller for managing the running instance of a "round". All of the control and logic that defines both the "King of the Hill" and "Assault" game modes would be contained in the scripted bindings of this *GameObject*. The *Level* doesn't really have any control about the actual layout of the *Level*, but objects that are used in the game mode register themselves with the *Level* to be tracked. The *Level* is also what controls the in-game options screen.

Figure 78: List of components attached to levels

GameManager

The *GameManager* tracks and maintains the number of current players, score, kills and other game data pertinent to the running instance.

LuaBinder

The *LuaBinder* is the main means for the developer to add the custom logic that a *Level* must execute in order to maintain and manage the game mode. Logic scripts would manage the following...

- When the game is over
- Scoring of the game
- When to pop up the options menu
- How to respawn a character

Appendix A — References

Audio Design Document

Overview:

Goal:

The overall design of the audio is built for one specific reason: giving the player as much information about their surroundings as possible. Sounds effects are often used in this fashion for games, but music is typically used to set a mood or to drape a setting. Because *Trigger Happy* is a frantic, first-person shooter game, it makes sense for the music to give the player as much information as well. Finally, voiceovers will be used to provide the players with information in a match rather than to move the player along a story path.

Music:

The music will be generated on the fly (see the “Research” section below) based upon a number of in-game variables. This will subconsciously help the player to understand and react to the current game state without the music becoming overdone or boring. By modifying the general feel of the music based upon the game state, the player will prepare for battle or recognize that there is no danger nearby without even realizing it. Music will be based upon the metal genre and will be specifically influenced by seminal band X-Japan, who have a remarkable talent to switch back and forth between power ballads and fast metal within the span of a song. A focus on a continual beat and a steady guitar line will be very important. Songs such as "Dahlia" and the 29 minute epic "Art of Life" are highlighted selections and influences from their discography.

Sound Effects:

The sound effects will also provide the player with information about the world, but this information will be more of a “recognize-and-respond” type than the background music’s subconscious effects. Sounds will not be 100% realistic, but will be overemphasized so that the player understands what they are and what is going on. Ambient effects will also be present in the game, but will simply convey localized information rather than game state information. They will help to add to the look and feel of the game.

Voice:

Voiceovers will explicitly be used to provide the player with information, such as how long until the round is over and other bits of the game state. Voiceovers will most of the time be heard through a loudspeaker sound system, so it can typically be assumed that all players can hear this information at the same time. These will be made very distinct so as not to interfere with player communication.

Music tone words: frenetic, metal, overemphasized, information-centric

Concepts:

- The audio must give the player as much information as possible about the game state as possible.
- The audio must not become so overwhelming that the player has the desire to turn it off.
- The audio must not “clash” with variable sound effects or music tracks.

- The music should provide the player with a unique soundscape that they will never hear twice.

Research:

A significant amount of research must be done for the music system of the game. XACT and the Singularity audio engine will manage the voice and sound effect implementations.

The music system will systematically piece together a number of non-linear tracks based upon what is going on in the game world. There will be three main phases: “calm”, “danger”, and “heat-of-battle”. In addition, certain elements may be modified based upon variables such as the player’s health, the proximity of the nearest enemy, and other factors.

A special audio system will be developed to take this into account and implement it. This system will determine the phase based upon constantly updating information and then begin to build music from there. To ensure that the musical snippets piece together properly, music will be built from the ground up: first, a rhythm section (drums followed by bass) will be chosen to fit together. From there, only a certain number of melody tracks can be chosen. Many melody lines will be added to the game in order to ensure that the music is not heard often and the player does not become sick of the music. Finally, supporting strings and synthesizers will be added to make the piece sound more full and provide extra information (a specific instrument used as a drone, for instance, may tell the player that his health is low.)

Technology tests will need to be implemented in order to determine the exact methods of building the audio piece. Ideally, a combination of custom software and XACT will be able to keep everything in beat and blend the various tracks together. If necessary, FMOD or Wwise will be used to support virtual voices to expand upon the number of channels on the sound card. A final fallback method would utilize a custom MIDI interpreter and player. In this case, each MIDI cue would apply to a specific set of samples.

Additional sounds that may be applied to the music include destructive interference of particular frequencies on particular tracks and other DSP effects. Everything will be kept in a single key, but it may also be possible to modulate each track up to another key.

Implementation:

Voiceovers will be recorded in Digidesign Pro Tools with voice actors. These tracks will be cleaned up and processed before being brought into the game. They will be played with Microsoft XACT and Singularity’s audio engine. Voiceovers will be exported as .wav files and may or may not be compressed in XACT depending on the quality necessary.

Likewise, sound effects will be played with XACT and Singularity’s audio engine. They will be recorded through Digidesign Pro Tools or acquired through licensed sources. These effects will be heavily modified in Apple Logic to sound appropriate in the game. Sound effects will be exported as .wav files and compressed using XACT.

Finally, music will be played through a custom music generator described in the Research section. Non-linear music pieces will be generated in Apple Logic and output into .wav files. These .wav files will be compressed in XACT to save space. These music pieces should ideally be blended together in a virtual channel on the computer before being output to the sound card, as it will be made from a number of unique tracks.

Each file will be saved as a standard LPCM 16-bit wav file at 44100kHz. These files will then be brought into XACT and compressed into ADPCM to save space in memory (with the exception of voiceovers, which will remain uncompressed). Techniques described here will be reconsidered if the research phase shows a significant use of memory. Filenames will begin with a lowercase prefix and will be followed by an underscore and a PascalCase descriptor.

The following prefixes will be utilized: menu, player, weapon, modifier, ambient, vo, music.

Overall Development / Details:

Music:

Music tracks will be created in Apple Logic and rerecorded live if time allows. These tracks will be made up of a number of individual pieces that will be strung together live by the adaptive music engine used by the game. Should this adaptive music engine fail or show itself to be unsuccessful, looping music tracks will be written in Apple Logic and potentially rerecorded live.

Sound:

Sound effects will be taken from a mix of original Foley sessions and modified licensed sound effects. These tracks will be overemphasized and unrealistic compared to their real-world counterparts, but will still be recognizable to players. These high-power effects will make players' adrenaline rush and allow them to consciously notice and acknowledge the information being conveyed.

Voice:

Voiceovers will be recorded in Digidesign Pro Tools and pre-processed to save CPU time in the game. These voiceovers will all be linear and will not be pieced together like the music. Three voice actors will be used: one adult female for the theme song, one adult male for all of the player sounds, and one adult male for the announcer.

Content List:

Sound effects:

Sample Asset Name. If starred, the source file may be the same as another.

Description	A description of what the asset sounds like.
Play cue	The event that causes the asset to play.
Source	The source of the asset. Synthesizers, foley sessions, or licensed and modified sound effects.
Length	How long the asset plays for. If starred, final

	length depends on animation lengths.
Loops	Does the asset loop? (Boolean)
3D positional	Is the asset positioned in 3D space? (Boolean -- yes means 3D, no means 2D)
Others can hear	Will others hear this particular effect in the game world? (Boolean. Some assets will play on all machines regardless.)
Filename	The source filename for the asset. Will not appear in the final game.
Wavebank	The wavebank the source file can be found in. Will appear in the final game.
Soundbank	The soundbank the file can be accessed from. Will appear in the final game.
Cue name	The cue name used to call the sound. Will typically be the same as the Filename but without extension.

Interface (menu_)

Menu Cursor Move

Description	The menu cursor movement sound is similar to the click of a small metal object (such as a bullet) hitting a solid object.
Play cue	The arrow keys are used on the main menu to change a selection.
Source	Licensed track or Foley
Length	0.1 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	menu_CursorMove.wav
Wavebank	MenuEffects.xwb
Soundbank	MenuEffects.xsb
Cue name	menu_CursorMove

Menu Select

Description	The selection sound in Trigger Happy main menus is similar to the basic pistol's firing sound.
Play cue	Something is selected in one of the menus.
Source	Licensed track or Foley
Length	0.2 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	menu_Select.wav
Wavebank	MenuEffects.xwb
Soundbank	MenuEffects.xsb
Cue name	menu_Select

Menu Cancel

Description	The cancel sound is similar to the modifier destroyed sound.
Play cue	The player backs out of a menu or cancels a selection.
Source	Licensed track or Foley
Length	0.2 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	menu_Cancel.wav
Wavebank	MenuEffects.xwb
Soundbank	MenuEffects.xsb
Cue name	menu_Cancel

Player (player_)

Player Effects

****Player-on-Player Collision***

Description	Player-on-player collision is an organic sound that will be recorded directly from flesh-on-flesh.
Play cue	The player accidentally bumps into another player in the heat of battle.
Source	Flesh-on-flesh collision (Foley)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	player_PlayerCollision.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_PlayerCollision

Player Hurt

Description	The player hurt sound is a male human sound of pain and grunting.
Play cue	The player takes damage and is not in a "danger" level of health.
Source	Human male grunt (Voice actor)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	player_Hurt.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_Hurt

Player Hurt (Near-Dying)

Description	The near-dying player hurt sound is a human male groaning in agony.
Play cue	The player takes damage and is in a "danger" level of health.
Source	Human male grunt (Voice actor)
Length	0.3 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	player_NearDying.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_NearDying

Player Respawn

Description	A whooshing sound generated by the release of a new clone.
Play cue	The player respawns on the level.
Source	Wind sounds (Foley)
Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	player_Respawn.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_Respawn

Player Health Recharge

Description	The player health recharge is a quiet drone ascending in pitch until the player's health is fully charged. The pitch is based solely upon how much health has been charged rather than what percentage of health the charge has reached.
Play cue	The sound will play when the player's health begins to recharge.
Source	Synthesizer
Length	3 seconds
Loops	Yes (until health is recharged)
3D positional	No
Others can hear	No
Filename	player_HealthRecharge.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_HealthRecharge

Footsteps

Description	The footsteps are recordings of humans walking/running *(on the following materials: stone, metal, carpet, wood, tile, rubber, and glass).
Play cue	These will play when the player walks on top of the respective in-game material.
Source	Shoes on materials (Foley)
Length	<0.1 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	player_Footsteps.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_Footsteps

Player Lands on Ground

Description	This sound is a louder, multi-layered version of the material's footstep recording. *(The materials are stone, metal, carpet, wood, tile, rubber, and glass).
Play cue	The player lands from a fall greater than N feet.
Source	Footsteps asset
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	player_Landing.wav
Wavebank	PlayerEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_Landing

Assault and King Of The Hill

FlagPpicked Up

Description	This is a celebratory signal to let the player know that he is doing the team a service. Similar to trumpet announcements in the courts of kings.
Play cue	The player picks up the flag to bring it to another checkpoint and is attacking.
Source	Synthesizer trumpets.
Length	0.75 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	player_FlagPickup
Wavebank	PlayerCTFEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_FlagPickup

Flag Dropped

Description	This is a tragic signal to let the player know that he failed. This is a variation of the trumpet announcement.
Play cue	The player drops the flag for whatever reason (death).
Source	Synthesizer trumpets.
Length	0.75 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	player_FlagDrop.wav
Wavebank	PlayerCTFEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_FlagDrop

Flag Brought to Checkpoint

Description	A variation on the flag picked up theme.This may become the same sound as the flag picked up.
Play cue	The player brings the flag to the checkpoint successfully.
Source	Synthesizer trumpets.
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	player_FlagCheckpoint.wav
Wavebank	PlayerAssaultEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_FlagCheckpoint

****Player Defends KOTH Spot***

Description	This is a soft hum that plays in the background as a notifier.
Play cue	The player is standing on the KOTH spot and is part of the team defending it. Stops playing when the player leaves the spot or the team loses control.
Source	Synthesizer or Foley
Length	2 seconds
Loops	Yes (indefinitely)
3D positional	No
Others can hear	No
Filename	player_KOTHDefending.wav
Wavebank	PlayerKOTHEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_KOTHDefending

Player Captures KOTH Spot

Description	This is the same success sound as the flag brought to the checkpoint.
Play cue	The player is on the KOTH spot when it is successfully captured.
Source	Synthesizer trumpets.
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	player_KOTHCapture.wav
Wavebank	PlayerKOTHEffects.xwb
Soundbank	PlayerEffects.xsb
Cue name	player_KOTHCapture

Weapons (weapon_)

Projectiles

Explosion

Description	A loud, high-impact explosion.
Play cue	An explosive item or projectile is set off.
Source	Fire and high impact (Licensed effects or Foley)
Length	0.75 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_Explosion.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_Explosion

Bullet Collision (Soft Material)

Description	The collision sound of a hard object against a soft material.
Play cue	A projectile hits a soft material.
Source	Material collisions (Foley)
Length	0.1 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_SoftCollision.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_SoftCollision

Bullet Collision (Hard Material)

Description	The collision sound of a hard object against a hard

	material.
Play cue	A projectile hits a hard material.
Source	Material collisions (Foley)
Length	0.1 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_HardCollision.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_HardCollision

Nullifier Gun

Switch to Nullifier Gun

Description	A short whoosh as the player pulls out the nullifier.
Play cue	The player switches to the Nullifer Gun.
Source	Wind sounds (Foley or licensed track)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_NullifierSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_NullifierSwitch

Nullifier Gun Fire

Description	A short, high-pitched whine with a number of harmonic frequencies attached.
Play cue	The player fires the Nullifier Gun.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_NullifierFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_NullifierFire

Nullifier Gun Reload

Description	A selection of short, quick sounds in succession while the player reloads.
Play cue	The player reloads the Nullifier Gun
Source	Foley or licensed tracks
Length	1 second

Loops	No
3D positional	No
Others can hear	No
Filename	weapon_NullifierReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_NullifierReload.wav

Sniper Rifle

Switch to Sniper Rifle

Description	A series of hard objects again flesh and cloth.
Play cue	The player switches to the Sniper Rifle.
Source	Misc. collisions (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_SniperSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_SniperSwitch

Sniper Rifle Fire

Description	The Sniper Rifle makes a loud, booming explosive sound from its barrel when it fires.
Play cue	The player fires the Sniper Rifle.
Source	Explosion asset and pistol asset.
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_SniperFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_SniperFire

Sniper Rifle Reload

Description	The sound of the player pushing a new bullet into the Sniper Rifle.
Play cue	The player reloads the Sniper Rifle.
Source	Mechanical sounds (Foley)
Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_SniperReload.wav
Wavebank	WeaponEffects.xwb

Soundbank	WeaponEffects.xsb
Cue name	weapon_SniperReload

Rocket Launcher

Switch to Rocket Launcher

Description	A series of hard objects again flesh and cloth. Possibly the same as the switch to the Sniper Rifle.
Play cue	The player switches to the Rocket Launcher.
Source	Misc. collisions (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_RocketSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_RocketSwitch

Rocket Launcher Fire

Description	A loud impact from the recoil.
Play cue	The player fires the Rocket Launcher.
Source	Collision sounds (Foley)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_RocketFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_RocketFire

Rocket Launcher Reload

Description	The sound of a player pulling a rocket out from a pouch and putting it into the machine.
Play cue	The player reloads the Rocket Launcher.
Source	A combination of the Sniper Rifle Reload and the Grenade Switch
Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_RocketReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_RocketReload

Assault Rifle

Switch to Assault Rifle

Description	A series of hard objects again flesh and cloth. Possibly the same as the switch to the Sniper Rifle.
Play cue	The player switches to the Assault Rifle.
Source	Collision sounds (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_AssaultSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_AssaultSwitch

Assault Rifle Fire

Description	A fast-paced firing sound with a slight bit of grinding.
Play cue	The player fires the Assault Rifle.
Source	Licensed tracks
Length	0.2 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_AssaultFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_AssaultFire

Assault Rifle Reload

Description	Fleshy objects against metal.
Play cue	The player reloads the Assault Rifle.
Source	Misc. collisions (Foley)
Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_AssaultReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_AssaultReload

Shotgun

Switch to Shotgun

Description	A series of hard objects again flesh and cloth. Possibly the same as the switch to the Sniper Rifle.

Play cue	The player switches to the Shotgun.
Source	Collision sounds (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_ShotgunSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_ShotgunSwitch

Shotgun Fire

Description	The sound of many small explosions at once and a short clicking afterwards (as is typical for a shotgun).
Play cue	The player fires the Shotgun.
Source	Modified Explosion asset
Length	0.5 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_ShotgunFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_ShotgunFire

Shotgun Reload

Description	The sound of three shotguns reloading at varying rates.
Play cue	The player reloads the Shotgun.
Source	Reloading (Licensed tracks)
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_ShotgunReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_ShotgunReload

Flamethrower

Switch to Flamethrower

Description	A series of hard objects again flesh and cloth. Possibly the same as the switch to the Sniper Rifle.
Play cue	The player switches to the Flamethrower.
Source	Collision sounds (Foley)
Length	0.3 seconds*

Loops	No
3D positional	No
Others can hear	No
Filename	weapon_FlameSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_FlameSwitch

Flamethrower Fire

Description	The high-pitched whine and sizzle of a microwave ray.
Play cue	The player fires the Flamethrower.
Source	Foley or licensed track
Length	0.5 seconds
Loops	Yes (until the player releases the fire button)
3D positional	Yes
Others can hear	Yes
Filename	weapon_FlameFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_FlameFire

Flamethrower Reload

Description	The sound of the player pulling out a new cartridge for the Flamethrower.
Play cue	The player reloads the Flamethrower.
Source	Foley or licensed track
Length	2 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_FlameReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_FlameReload

Pistol

Switch to Pistol

Description	A series of hard objects again flesh and cloth. Possibly the same as the switch to the Sniper Rifle.
Play cue	The player switches to the Pistol.
Source	Collision sounds (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_PistolSwitch.wav

Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_PistolReload

Pistol Fire

Description	A small, high-pitched pop.
Play cue	The player fires the Pistol.
Source	Balloon popping (Foley)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_PistolFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_PistolFire

Pistol Charging

Description	A high-pitched whine from low to high.
Play cue	The player charges the Pistol.
Source	Modified flamethrower sound.
Length	2.5 seconds*
Loops	Yes (until the player releases the fire button)
3D positional	Yes
Others can hear	Yes
Filename	weapon_PistolCharge.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_PistolCharge

Pistol Charged Fire

Description	A louder, mid-pitched pop.
Play cue	The player fires the charged Pistol.
Source	Balloon popping (Foley)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_PistolFireCharged.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_PistolFireCharged

Pistol Reload

Description	The sound of ping-pong balls being pushed into a small container.
Play cue	The player reloads the Pistol.
Source	Balls into containers (Foley)

Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_PistolReload.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_PistolReload.wav

Grenades

Switch to Grenades

Description	The sound of somebody pulling a large object out of a pouch.
Play cue	The player switches to Grenades.
Source	Hard objects and materials (Foley)
Length	0.3 seconds*
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_GrenadeSwitch.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_GrenadeSwitch

Lob Grenade

Description	The pop of a soda tab and then a whoosh as it flies through the air.
Play cue	The player fires a grenade.
Source	Soda can and wind (Foley)
Length	2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	weapon_GrenadeFire.wav
Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_GrenadeFire

Grenade Reload

Description	A small object moving against cloth.
Play cue	The player reloads a new grenade.
Source	Collision and cloth (Foley)
Length	1 second
Loops	No
3D positional	No
Others can hear	No
Filename	weapon_GrenadeReload.wav

Wavebank	WeaponEffects.xwb
Soundbank	WeaponEffects.xsb
Cue name	weapon_GrenadeReload

Modifiers (modifer_)

General

Reticle Alert

Description	A short alarm to alert the player of the reticle
Play cue	The player moves his cursor over a modifier in play that has somebody within its proximity.
Source	Modified ambient alarm asset
Length	0.3 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_Alert.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_Alert

Modifier Charge

Description	A high-pitched whine.
Play cue	The player charges a modifier in order to use a higher-level version.
Source	Modified pistol charge
Length	0.5 seconds
Loops	Yes
3D positional	Yes
Others can hear	Yes
Filename	modifier_Charge.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_Charge

Modifier Deploy

Description	A mechanical movement sound as the modifier deploys.
Play cue	The sound played when a modifier becomes active.
Source	Mechanical objects (Foley)
Length	0.2 seconds*
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_Deploy.wav
Wavebank	ModifierEffects.xwb

Soundbank	ModifierEffects.xsb
Cue name	modifier_Deploy

Modifier Shot

Description	A bullet collision with an emphasized solid sound.
Play cue	The player damages a modifier.
Source	Modified bullet collision asset
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_Shot.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_Shot

Modifier Destroyed

Description	A small explosion.
Play cue	A modifier is destroyed.
Source	Modified explosion sound
Length	0.5 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_Destroy.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_Destroy

Inverse Gravity

Inverse Gravity Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a piano when the modifier cartridge is inserted.
Play cue	The Inverse Gravity modifier is selected.
Source	Synthesizer and Foley.
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_InverseSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_InverseSelect

Inverse Gravity Proximity

Description	A quick ascending series of notes using a piano. Major key.
Play cue	The player walks within the proximity radius of an Inverse Gravity modifier.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_InverseProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_InverseProximity

Inverse Gravity Trigger

Description	A quick, loud ascending series of notes using a piano. Minor key.
Play cue	The Inverse Gravity modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_InverseTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_InverseTrigger

Increase Gravity

Increase Gravity Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a harpsichord when the modifier cartridge is inserted.
Play cue	The Increase Gravity modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_IncreaseSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IncreaseSelect

Increase Gravity Proximity

Description	A quick ascending series of notes using a harpsichord. Major key.
Play cue	The player walks within the proximity radius of an Increase Gravity modifier.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_IncreaseProximity
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IncreaseProximity

Increase Gravity Trigger

Description	A quick, loud ascending series of notes using a harpsichord. Minor key.
Play cue	The Increase Gravity modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_IncreaseTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IncreaseTrigger

Knockback

Knockback Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a vibraphone when the modifier cartridge is inserted.
Play cue	The Knockback modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_KnockbackSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_KnockbackSelect

Knockback Proximity

Description	A quick ascending series of notes using a

	vibraphone. Major key.
Play cue	The player walks within the proximity radius of a Knockback modifier.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_KnockbackProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_KnockbackProximity

Knockback Trigger

Description	A quick, loud ascending series of notes using a vibraphone. Minor key.
Play cue	The Knockback modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_KnockbackTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_KnockbackTrigger

Accelerator

Accelerator Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a piccolo when the modifier cartridge is inserted.
Play cue	The Accelerator modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_AcceleratorSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_AcceleratorSelect

Accelerator Proximity

Description	A quick ascending series of notes using a piccolo. Major key.

Play cue	The player walks within the proximity radius of an Accelerator modifier.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_AcceleratorProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_AcceleratorProximity

Accelerator Trigger

Description	A quick, loud ascending series of notes using a piccolo. Minor key.
Play cue	The Accelerator modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_AcceleratorTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_AcceleratorTrigger

Barrier

Barrier Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a tuba when the modifier cartridge is inserted.
Play cue	The Barrier modifier is selected.
Source	Synthesizer
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_BarrierSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_BarrierSelect

Barrier Proximity

Description	A quick ascending series of notes using a tuba. Major key.
Play cue	The player walks within the proximity radius of a

	Barrier modifier.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_BarrierProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_BarrierProximity

Barrier Trigger

Description	A quick, loud ascending series of notes using a tuba. Minor key.
Play cue	The Barrier modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_BarrierTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_BarrierTrigger

Illusion

Illusion Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a string quartet when the modifier cartridge is inserted.
Play cue	The Illusion modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_IllusionSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IllusionSelect

Illusion Proximity

Description	A quick ascending series of notes using a string quartet. Major key.
Play cue	The player walks within the proximity radius of an Illusion modifier.

Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_IllusionProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IllusionProximity

Illusion Trigger

Description	A quick, loud ascending series of notes using a string quartet. Minor key.
Play cue	The Illusion modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_IllusionTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_IllusionTrigger

Grow

Grow Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a male choir when the modifier cartridge is inserted.
Play cue	The Grow modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_GrowSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_GrowSelect

Grow Proximity

Description	A quick ascending series of notes using a male choir. Major key.
Play cue	The player walks within the proximity radius of a Grow modifier.
Source	Synthesizer

Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_GrowProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_GrowProximity

Grow Trigger

Description	A quick, loud ascending series of notes using a male choir. Minor key.
Play cue	The Grow modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_GrowTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_GrowTrigger

Shrink

Shrink Select

Description	A short whoosh and metal against material as the player pulls out the modifier gun. A single note using a female choir when the modifier cartridge is inserted.
Play cue	The Shrink modifier is selected.
Source	Synthesizer and Foley
Length	0.5 seconds
Loops	No
3D positional	No
Others can hear	No
Filename	modifier_ShrinkSelect.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_ShrinkSelect

Shrink Proximity

Description	A quick ascending series of notes using a female choir. Major key.
Play cue	The player walks within the proximity radius of a Shrink modifier.
Source	Synthesizer
Length	0.2 seconds

Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_ShrinkProximity.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_ShrinkProximity

Shrink Trigger

Description	A quick, loud ascending series of notes using a female choir. Minor key.
Play cue	The Shrink modifier is triggered.
Source	Synthesizer
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	Yes
Filename	modifier_ShrinkTrigger.wav
Wavebank	ModifierEffects.xwb
Soundbank	ModifierEffects.xsb
Cue name	modifier_ShrinkTrigger

Ambient (ambient_)

Conveyor Belts and Sorting Machines

Description	This is the sound of machinery running.
Play cue	The conveyor belts and sorting machines play when the player enters the listening radius if the machines are on.
Source	Modified machine sounds (Licensed effect)
Length	12 seconds
Loops	Yes
3D positional	Yes
Others can hear	No
Filename	ambient_Machine.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Machine

***"Muzak"**

Description	Old muzak playing in the museum. Has a jazz-influenced feel.
Play cue	When this plays, the background music will duck slightly to emphasize the sound, much like all voiceovers.
Source	Logic
Length	~20 seconds
Loops	Yes

3D positional	Yes
Others can hear	No
Filename	ambient_Muzak.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Muzak

****Experiments***

Description	A collection of strange sounds from odd experiments.
Play cue	Plays when the player is in the research lab.
Source	Foley or licensed tracks (or some mix thereof)
Length	10 seconds
Loops	Yes
3D positional	Yes
Others can hear	No
Filename	ambient_Experiments.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Experiments.wav

Folding Sound for Equipment

Description	Mechanical sound similar to the conveyor belts and machines, but not looping.
Play cue	The equipment fold is triggered.
Source	Machinery (Licensed and modified track).
Length	2 seconds*
Loops	No
3D positional	Yes
Others can hear	No
Filename	ambient_Fold.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Fold

WarningSirens

Description	A klaxon sound.
Play cue	Plays when an alarm is going off.
Source	Siren or alarm (Licensed sound or Foley)
Length	2 seconds
Loops	Yes
3D positional	Yes
Others can hear	No
Filename	ambient_Alarm.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Alarm

Warning Lights Flash

Description	A slightly electric sound much like that of a flickering light bulb. Bzzt.
Play cue	Plays whenever a warning light turns on.
Source	Electricity (Licensed sound or Foley)
Length	0.2 seconds
Loops	No
3D positional	Yes
Others can hear	No
Filename	ambient_Lights.wav
Wavebank	AmbientEffects.xwb
Soundbank	AmbientEffects.xsb
Cue name	ambient_Lights

Voiceovers:

"We have a new Valedictorian."

Description	This voiceover will be processed to sound as if it is coming out of a middle-quality loudspeaker.
Play cue	This will be played whenever a person ascends from second place to first place.
Source	Male voice actor
Length	Depends on voice actor.
Voice Actor	Male II
Loops	No
3D positional	No
Others can hear	No
Filename	vo_Valedictorian.wav
Wavebank	Voiceovers.xwb
Soundbank	Voiceovers.xsb
Cue name	vo_Valedicatorian

"The IP has captured a hill."

Description	This voiceover will be processed to sound as if it is coming out of a middle-quality loudspeaker.
Play cue	This will be played whenever the IP captures a hill.
Source	Male voice actor
Length	Depends on voice actor.
Voice Actor	Male II
Loops	No
3D positional	No
Others can hear	No
Filename	vo_HillIP.wav
Wavebank	Voiceovers.xwb
Soundbank	Voiceovers.xsb
Cue name	vo_HillIP

"The ACF has captured a hill."

Description	This voiceover will be processed to sound as if it is coming out of a middle-quality loudspeaker.
Play cue	This will be played whenever the ACF captures a hill.
Source	Male voice actor
Length	Depends on voice actor.
Voice Actor	Male II
Loops	No
3D positional	No
Others can hear	No
Filename	vo_HillACF.wav
Wavebank	Voiceovers.xwb
Soundbank	Voiceovers.xsb
Cue name	vo_HillACF

"The IP has reached a checkpoint."

Description	This voiceover will be processed to sound as if it is coming out of a middle-quality loudspeaker.
Play cue	This will be played whenever the IP brings the flag to a checkpoint.
Source	Male voice actor
Length	Depends on voice actor.
Voice Actor	Male II
Loops	No
3D positional	No
Others can hear	No
Filename	vo_CheckpointIP.wav
Wavebank	Voiceovers.xwb
Soundbank	Voiceovers.xsb
Cue name	vo_CheckpointIP

"The ACF has reached a checkpoint."

Description	This voiceover will be processed to sound as if it is coming out of a middle-quality loudspeaker.
Play cue	This will be played whenever the ACF brings a flag to a checkpoint.
Source	Male voice actor
Length	Depends on voice actor.
Voice Actor	Male II
Loops	No
3D positional	No
Others can hear	No
Filename	vo_CheckpointACF.wav
Wavebank	Voiceovers.xwb
Soundbank	Voiceovers.xsb
Cue name	vo_CheckpointACF

Background Music:

The full extent of the construction and stitching of these pieces will depend on the research project and will be further fleshed out later.

Description	See each piece description
Play cue	When the MusicManager dictates
Source	Logic
Length	1-4 measures
Loops	No (but seams connect)
Filename	music_[Instrument].wav
Wavebank	NonLinearMusic.xwb
Soundbank	NonLinearMusic.xsb
Cue name	music_Level

Drum Pieces

These drum pieces will use a heavy rock kit. The drum pieces will set the rhythm and the tempo for the track, and other pieces will be chosen based upon the drum.

Bass Pieces

The bass pieces will be chosen based upon the drum track and will utilize a rhythm that blends intricately with the drum pieces. Bass lines will be relatively complex compared to the rhythm guitar lines. The drum pieces and the bass pieces are considered the tight rhythm section, though the rhythm guitar could also be considered part of this section.

Rhythm Guitar Pieces

The rhythm guitar lines will be chosen based upon the drum rhythm and the bass rhythm / chord structure. Rhythm guitar lines will be relatively simple and typically utilize major chord structures.

Lead Guitar Pieces

The lead guitar lines will be chosen based solely upon the rhythm guitar section and will include both melody lines and dual harmony solos.

String Pieces

String pieces will support the guitar pieces and help to soften the full piece of music. Strings, much like synths, will not always play and will often be whole notes in the background.

Synth Pieces

Synthesizer pieces will not always play, and will often act as drones. Occasionally, the synthesizer may be brought in to act as melody or as harmony to the lead guitar.

Piano Pieces

The piano pieces are used when the player dies. The music will begin to fade into a piano line similar to the current melody, and will fade back into the full track when the player respawns.

Misc. Pieces

Miscellaneous instruments such as gongs fit in here.

NOTE: Background music in Trigger Happy will be developed as a series of smaller pieces broken down by instrument and designed to piece together fluidly. Pieces are described here. The overall system is described in the “Research” section above. In addition, “fallback” music tracks have been described should the adaptive system fail.

Trigger Happy theme

Description	The Trigger Happy theme will be played on the main menu screen when the game is loaded. It will be a lower-energy track utilizing the same instruments as the rest of the game's soundtrack. There will be female lead singer and a fully developed set of lyrics. This track will have an especially catchy introduction to hook the player and make the game immediately more appealing upon load. The best-case scenario is a player who listens fully to the track each time. This track should be about three minutes long.
Play cue	When the player is on the main menu.
Source	Logic
Length	180 seconds
Loops	No
Filename	music_Theme.wav
Wavebank	LinearMusic.xwb
Soundbank	LinearMusic.xsb
Cue name	music_Theme

Fallback Level Background

Description	This track is a high-energy masterwork of metal. A double-kick drum will become a prominent piece of the music, as will a technical guitar melody line. Heavy rhythm guitars will be doubled up to thicken the sound, and the key of the music will be a minor key. This track will give players the sense of battle and adrenaline.
Play cue	When the player is in the level.
Source	Logic
Length	300 seconds*
Loops	Yes
Filename	music_Level.wav
Wavebank	LinearMusic.xwb
Soundbank	LinearMusic.xsb
Cue name	music_Level

Death Soundtrack

Description	A short piano track will be played as the respawn counter counts down. When the player respawns, the level background will begin to play again.
Play cue	When the player dies.
Source	Logic
Length	15 seconds*
Loops	No
Filename	music_Death.wav
Wavebank	LinearMusic.xwb
Soundbank	LinearMusic.xsb

Cue name	music_Death

Game Production Document

Summary

In order to maintain a sense of order in many facets of our development, a system for organizing files, assets, and even code must be established. We have written this production guide to help maintain this order and allow for a more cohesive development experience for the development team. By maintaining the coding standards established in this document, we can ensure that team members can stay up to speed on other team member's work. We can also ensure that should gaps need to be filled in development, these production standards can help maintain an understanding of where team members are currently at in their development timelines.

Document Scope

In this document, we have detailed out the various aspects of production organization. This includes naming conventions and folder organizations for all files pertaining to *Trigger Happy* and organizational styles for writing code and comments. The document is organized into tables that detail out these specific tasks and describe how the convention works.

FIle Naming Convention

Type	Standard/Convention	Example
Texture	PascalCase naming with .png or .tga extension. Lowercase prefix documenting the context of use.	`Room_Box.png`
Model	PascalCase naming with .fbx extension. Lowercase prefix documenting the context of use.	`Room_Box.fbx`
Effects	PascalCase naming with .fx extension	`BasicEffect.fx`
Lua Scripts	PascalCase naming with .lua extension	`MainFunction.lua`
Raw Audio Resources	Lowercase prefix followed by an underscore followed by a PascalCase name. See Audio Design Document for list of prefixes. Contains a .wav extension.	`menu_CursorMove.wav`
XACT Settings	PascalCase name with .xgs extension.	`TriggerHappy.xgs`
SoundBank	PascalCase name with .xsb extension.	`MenuEffects.xsb`
WaveBank	PascalCase name with .xwb extension.	`MenuEffects.xwb`
Cue Names	Lowercase prefix followed by an underscore followed by a PascalCase name. See Audio	`menu_CursorMove`

	Design Document for list of prefixes. No extension.	

Document Structure

Path	Description
.\	
assets\	Asset files used in the development of the game
audio\	XACT audio files (*.xgs, *.xsb, *.xwb)
effects\	Effect files(*.fx)
models\	Model files(*.fbx)
<maya structure>	The Maya texture and model hierarchy
textures\	Texture files(*.png, *.jpg)
materials\	General uncustomized surfaces
<room>\	Customized, room specific textures
reference\	Reference images
scripts\	Script files(*.lua)
code\	Engine\Component source code files
include\	Header files(*.h*.inc)
src\	Source files(*.cpp)
lib\	Library files(*.lib)
scripts\	Project files(*.vcproj)
tools\	External tools or projects
documentation\	Documentation files

Coding Standards

Naming Conventions

Type	Standard\Convention	Example
Namespaces	Pascal Case, no underscores. Use Singularity.{ComponentType} as root.	`Singularity::Graphics` `Singularity::Physics`

Classes and Structs	Pascal Case, no underscores or leading "C" or "cls". Classes may begin with an "I" only if the letter following the I is not capitalized, otherwise it looks like an Interface. Classes should not have the same name as the namespace in which they reside. Any acronyms of three or more letters should be pascal case, not all caps. Try to avoid abbreviations, and try to always use nouns.	`GraphicsDevice`
		`XmlLoader`
		`AnimationState`

Exception Classes	Follow class naming conventions, but add Exception to the end of the name	`InvalidInputException`
Interfaces	Follow class naming conventions, but start the name with "I" and capitalize the letter following the "I"	`IGraphicsDevice`
Enumerations	Follow class naming conventions. Do not add "Enum" to the end of the enumeration name. If the enumeration represents a set of bitwise flags, end the name with a plural.	`SchedulingOptions`
Functions	Pascal Case, no underscores, try to avoid abbreviations.	`void DoSomething(...)`
Properties	Follow Function naming conventions. Accessors should start with "Get_" and modifiers with "Set_"	`int Get_Something()` `void Set_Something(int value)`
Parameters	Camel Case. Try to avoid abbreviations.	`int deviceType`
Procedure-Level Variables	Camel Case. Try to avoid abbreviations.	`unsigned indexType`
Class-Level Private and Protected Variables	Camel Case with Leading "m_". Use standard type naming notation(i for ints, f for floats, etc.). Do not use single character variable names like i, j or k, unless that name is the commonly	`unsigned m_iIndex`

	accepted mathematical notation.	
Class-Level Global Variables	Follow Class-Level naming convention. Leading "g_" instead of the standard "m_"	`void* g_pInstances`

Comments

- All files should start with a file comment in the style:

```
/*
 * File.h
 *
 * Overview of what is in this file, why it exists and how it fits
 * into "The Big Picture".
 *
 * Author: Your Names
 *
 * Date: <Date of Origination>
 */
```

Class Syntax

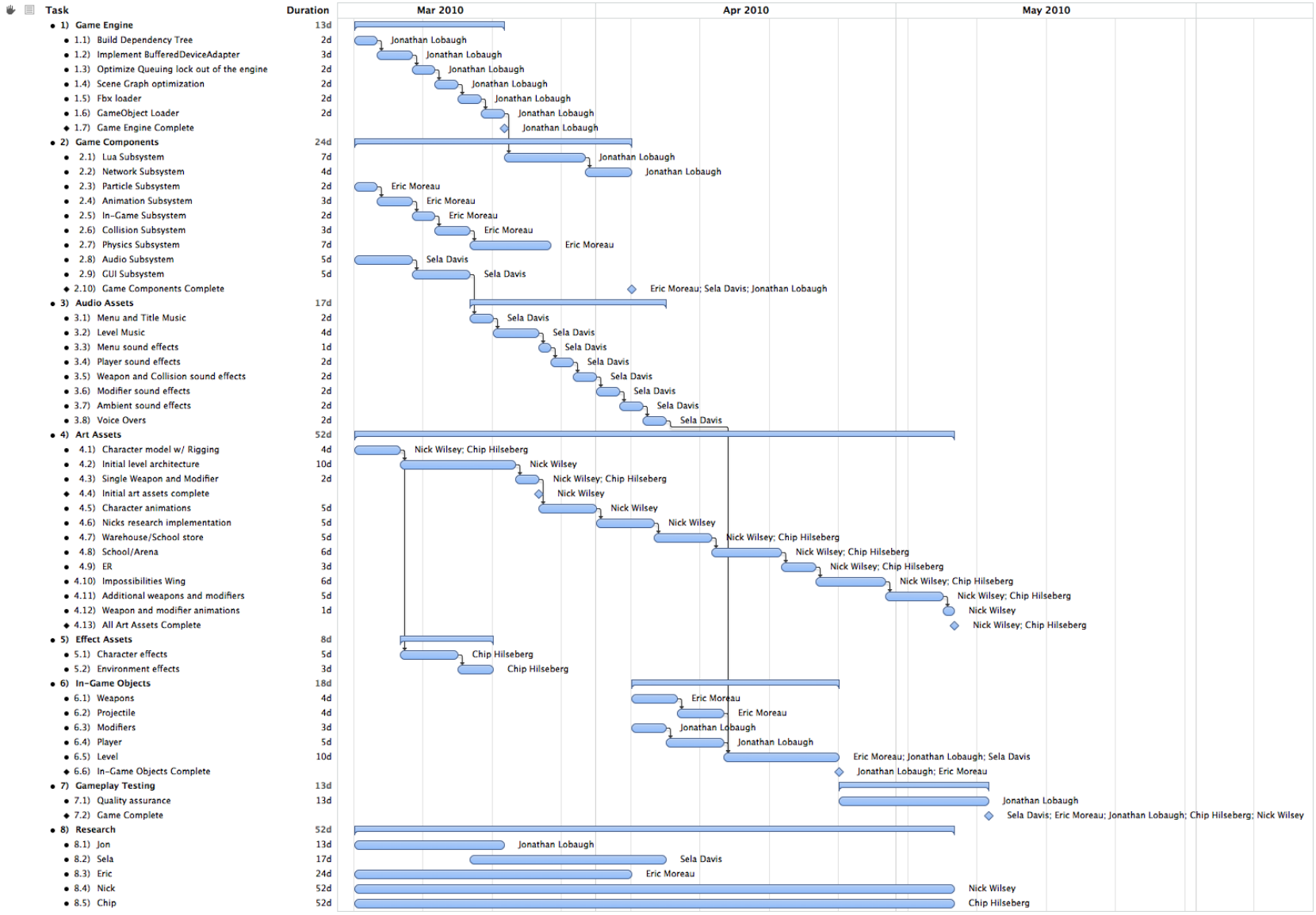
```
class Class_One: public Class_Parent
{
   private:
      #pragma region Variables
      // variables
      #pragma endregion

   Properties:
      #pragma region Methods
      // methods
      #pragma endregion

   public:
      #pragma region Properties
      // accessors and mutators
      #pragma endregion

      #pragma region Constructors and Finalizers
      // constructors and deconstructors
      #pragma endregion
};
```

File format

- Only one class (and any associated typedefs, constants, etc.) should be declared in each .h/.cpp pair.
- Files should be named for the (singular) class that they contain, i.e. Class_Name.h/Class_Name.cpp

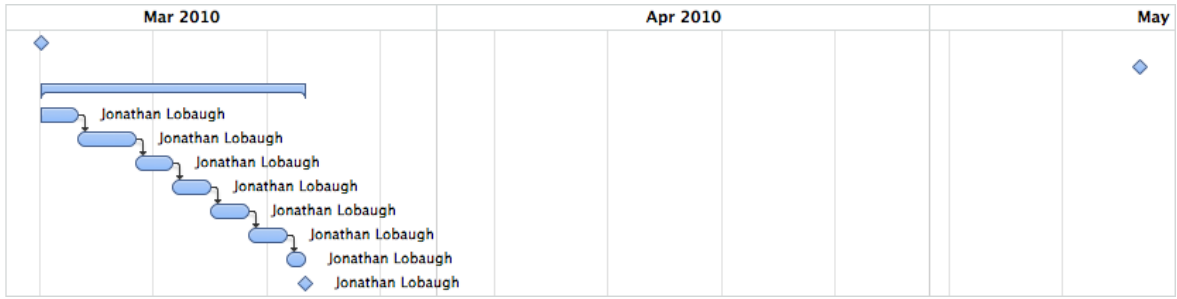
Development Timeline

Overall Timeline



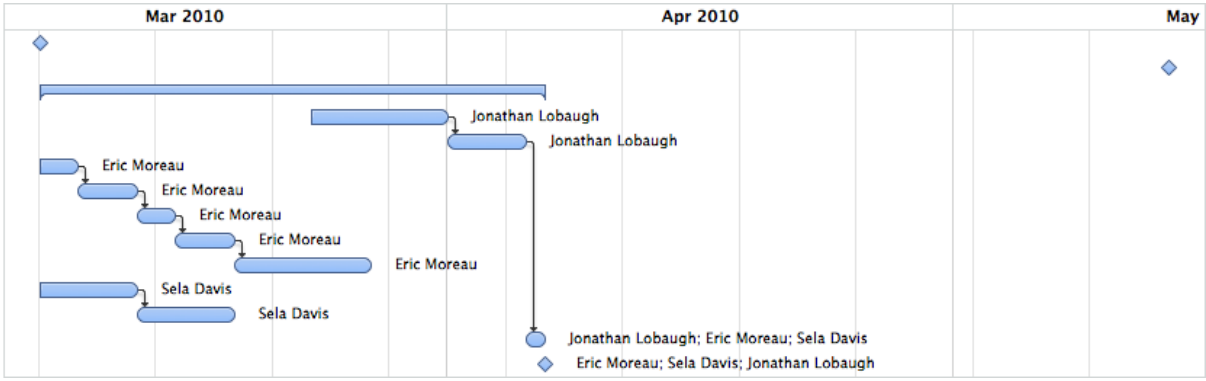
Game Engine Timeline

Task	Duration	Start	End
1) Start Development		3/8/10	3/8/10
2) Game Complete		5/14/10	5/14/10
3) Game Engine	14d	**3/8/10**	3/24/10
3.1) Build Dependency Tree	2d	3/8/10	3/9/10
3.2) Implement BufferedDeviceAdapter	3d	3/10/10	3/12/10
3.3) Optimize Queuing	2d	3/13/10	3/15/10
3.4) Scene Graph optimization	2d	3/16/10	3/17/10
3.5) Fbx loader	2d	3/18/10	3/19/10
3.6) GameObject Loader	2d	3/20/10	3/22/10
3.7) Update Documentation	1d	3/23/10	3/23/10
3.8) Game Engine Complete		3/24/10	3/24/10



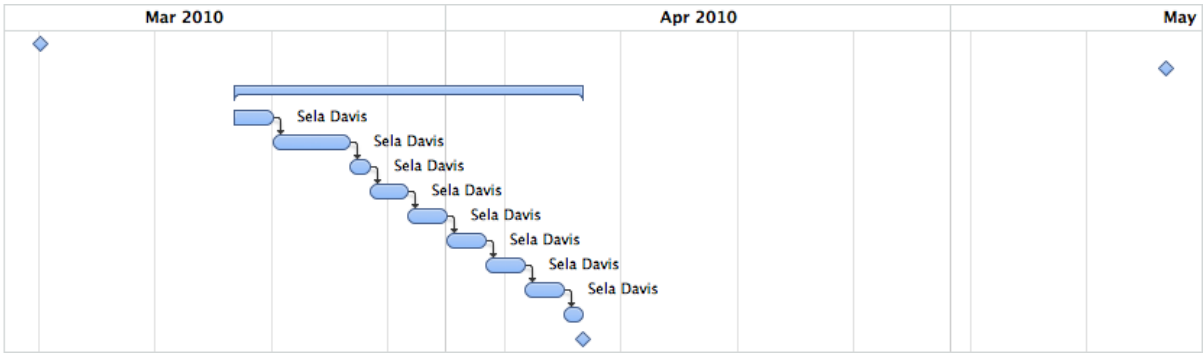
Game Components Timeline

Task	Duration	Start	End
1) Start Development		3/8/10	3/8/10
2) Game Complete		5/14/10	5/14/10
3) Game Components	26d	**3/8/10**	4/7/10
3.1) Lua Subsystem	7d	3/24/10	3/31/10
3.2) Network Subsystem	4d	4/1/10	4/5/10
3.3) Particle Subsystem	2d	3/8/10	3/9/10
3.4) Animation Subsystem	3d	3/10/10	3/12/10
3.5) In-Game Subsystem	2d	3/13/10	3/15/10
3.6) Collision Subsystem	3d	3/16/10	3/18/10
3.7) Physics Subsystem	7d	3/19/10	3/26/10
3.8) Audio Subsystem	5d	3/8/10	3/12/10
3.9) GUI Subsystem	5d	3/13/10	3/18/10
3.10) Update Documentation	1d	4/6/10	4/6/10
3.11) Game Components Complete		4/7/10	4/7/10



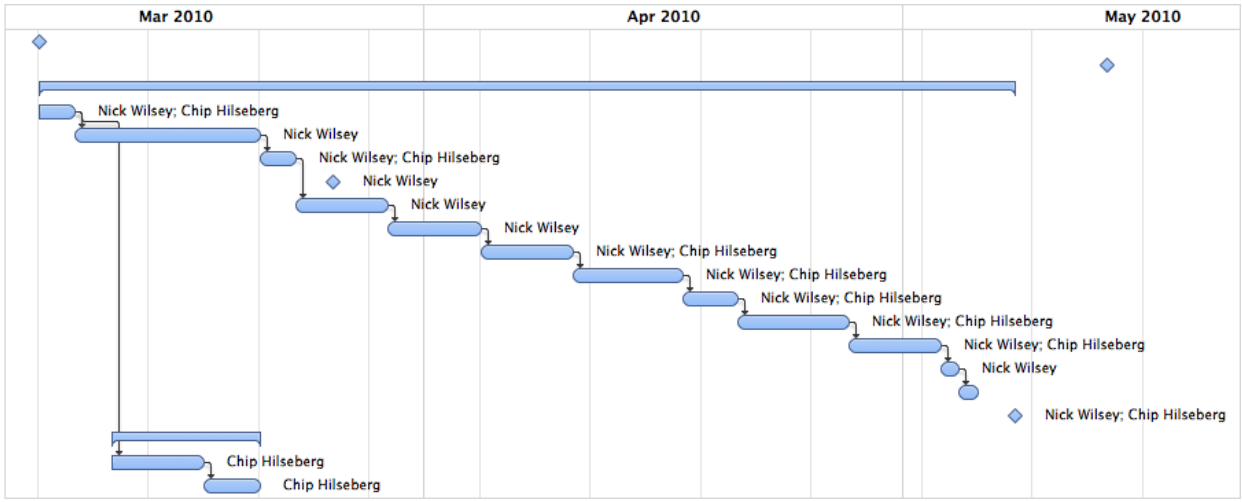
Audio Assets Timeline

Task	Duration	Start	End
1) Start Development		3/8/10	3/8/10
2) Game Complete		5/14/10	5/14/10
3) Audio Assets	18d	**3/19/...**	4/9/10
3.1) Menu and Title Music	2d	3/19/10	3/20/10
3.2) Level Music	4d	3/22/10	3/25/10
3.3) Menu sound effects	1d	3/26/10	3/26/10
3.4) Player sound effects	2d	3/27/10	3/29/10
3.5) Weapon and Collision sound effects	2d	3/30/10	3/31/10
3.6) Modifier sound effects	2d	4/1/10	4/2/10
3.7) Ambient sound effects	2d	4/3/10	4/5/10
3.8) Voice Overs	2d	4/6/10	4/7/10
3.9) Update Documentation	1d	4/8/10	4/8/10
3.10) Audio Assets Complete		4/9/10	4/9/10



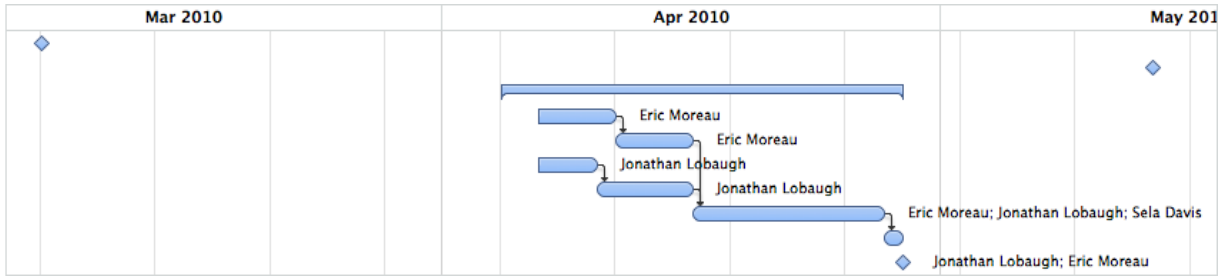
Art Assets Timeline

Task	Duration	Start	End
1) Start Development		3/8/10	3/8/10
2) Game Complete		5/14/10	5/14/10
3) Art Assets	53d	**3/8/10**	5/8/10
3.1) Character model w/ Rigging	2d	3/8/10	3/9/10
3.2) Initial level architecture	10d	3/10/10	3/20/10
3.3) Single Weapon and Modifier	2d	3/22/10	3/23/10
3.4) Initial art assets complete		3/26/10	3/26/10
3.5) Character animations	5d	3/24/10	3/29/10
3.6) Nicks research implementation	5d	3/30/10	4/3/10
3.7) Warehouse/School store	5d	4/5/10	4/9/10
3.8) School/Arena	6d	4/10/10	4/16/10
3.9) ER	3d	4/17/10	4/20/10
3.10) Impossibilities Wing	6d	4/21/10	4/27/10
3.11) Additional weapons and modifiers	5d	4/28/10	5/3/10
3.12) Weapon and modifier animations	1d	5/4/10	5/4/10
3.13) Update Documentation	1d	5/5/10	5/5/10
3.14) All Art Assets Complete		5/8/10	5/8/10
4) Effect Assets	8d	**3/12/...**	3/20/10
4.1) Character effects	5d	3/12/10	3/17/10
4.2) Environment effects	3d	3/18/10	3/20/10

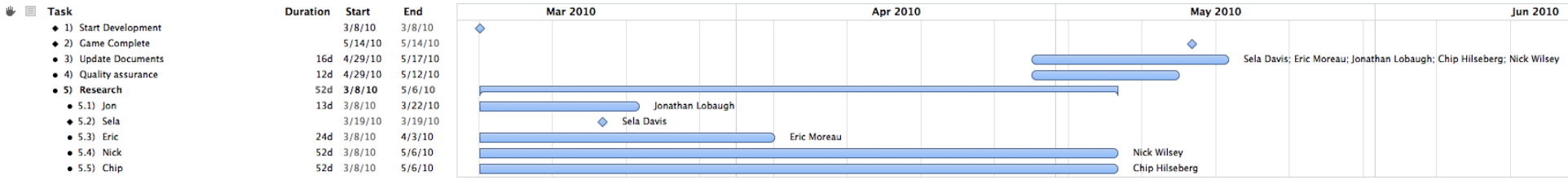


In-Game Objects Timeline

Task	Duration	Start	End
1) Start Development		3/8/10	3/8/10
2) Game Complete		5/14/10	5/14/10
3) In-Game Objects	21d	**4/5/10**	4/29/10
3.1) Weapons	4d	4/7/10	4/10/10
3.2) Projectile	4d	4/12/10	4/15/10
3.3) Modifiers	3d	4/7/10	4/9/10
3.4) Player	5d	4/10/10	4/15/10
3.5) Level	10d	4/16/10	4/27/10
3.6) Update Documentation	1d	4/28/10	4/28/10
3.7) In-Game Objects Complete		4/29/10	4/29/10



Miscellaneous Items Timeline



Audio Stuff By Sela

Capstone Research By Sela Davis

1. Quick Bio

1.1 Short Summary

Audio engine programmer, nonlinear composer for games ('audio manipulator'), metalsmith. MS Game Design & Development at RIT. Co-coordinator of IGDA Rochester.

1.2 Further Detail

Sela Davis is currently pursuing an M.S. in Game Design & Development at the Rochester Institute of Technology. She is an experience multidisciplinary worker, and thrives in such cultures. Her particular interests lie in artistic media -- particularly game audio, her specialty, and in metal, with which she is formally trained. She also enjoys writing for interactive media (such as ARGs!) and teaching others.

Sela is a published poet in Rochester, NY (under a psuedonym), a shown artist in San Francisco, CA, and has co-hosted three years worth of BarCampRochester. She is currently the co-coordinator and co-founder of IGDA Rochester along with Alex Lifschitz. She made her first appearance at San Francisco's GDC in 2009 flaunting hand-made etched copper business cards and a drive strong enough to melt steel. Sela will act as a Conference Associate at the upcoming GDC 2010 and as a member of the steering committee of a yet-unannounced new game industry conference.

1.3. CV / Resume

See attached document.

2. Problem and Solution

2.1 Overview

Music in long-play games such as first person shooters can become monotonous over time. Players have a tendency to turn off the music and/or all of the game's audio, which poses a particular problem for the game design and the audio content developers. One solution, as seen in games such as Team Fortress 2, is to avoid playing music entirely. This is not an optimal solution. An adaptive audio system is proposed for two reasons: to give more information to players about the state of the game world and to keep the music from becoming repetitive and monotonous.

There are a few major types of repetitive music. Linear music provides one: music becomes monotonous because it is not affected by the state of the world in any way. This type of music does not adapt to a game's situation and can often break immersion by syncing improperly to the player's emotions. This is analogous to playing the "Raider's March" over the Star Wars scene depicting Luke's family's corpses. Moreover, this music is typically a loop that plays over and over without variation.

Instead of this technique, many games use a set of triggers to change the music based upon the mood the player should be feeling. The flaw with this technique is the lack of blending from track to track (if the player can make a choice that changes the music, it jerks rather than flows) and the fact that the music typically has a set start point. This is a better solution, but still not an ideal one. There may be variation from area to area, but not from track to track. Again, once in these areas a track will loop constantly. The player may hear more tracks than in the previous style, but each is ultimately a piece of linear music that does not change over time.

The proposed solution takes this and expands it further. By using small pieces of music and generating combinations on the fly, nonlinear music can be used to provide a non-looping, non-repeating background track. Moreover, these tracks can be easily switched from one set of pieces to another based upon the state of the game world without a stop or a fade. The extra benefit of this system is the fact that it provides extra game state information to the player. Over time, the player should be able to understand subconscious what certain musical cues are. For instance, an increase in the tempo of the music may imply that a fight is beginning to break out in the game. Dynamically piecing together this information allows audio to move past individual linear music cues for determining the emotion of the player.

2.2 Relevance to Game

2.2.1 Problem to Game

Trigger Happy is a deathmatch-based First Person Shooter, or FPS. This style of game typically invites a long-term investment of time, during which traditional linear music often becomes boring to the player. Thousands of hours of linear music could, of course, be generated, but the resources necessary are costly and the required storage and memory space become quite large. Traditionally, FPSes and similar multiplayer games tend to avoid the use of music entirely, but there is so much more that can be done. FPSes in particular lend themselves well to adaptive audio, as there is often enough occurring in the world that most players have a difficult time determining their situation and danger level.

When players turn off their speakers, they lose important data. Hearing is one of the three (as of now) senses players use when they interact with games, and significant information can be lost if that sense is removed. This does not mean that the game is unplayable without sound; however, a significant amount of the atmosphere is lost without it. Even the loss of either music or sound effects is noticeable: music helps to create mood and emotion, and sound effects often provide world data.

2.2.2 Game to Problem

Trigger Happy is the perfect place to attempt an implementation for an adaptive audio system. This type of system helps to avoid the repetition found in long-term multiplayer games such as FPSes and MMOs, and has been implemented in games such as Spore and Slipgate Ironworks' unannounced MMO. By doing this, players are less likely to turn off the audio – the most significant problem game audio faces today – and are more likely to stay immersed in the game.

Moreover, a game like *Trigger Happy* has enough happening in the world that some players may find it difficult to understand exactly what is going on. While 3d positional audio provides many of these clues – and clues to events happening outside the player's point of view – players who turn off the audio miss all of this. As a team-based game, *Trigger Happy* requires players to know what is happening as much as possible. An adaptive music system like the one proposed allows players to identify the general

state of the game world through the use of various themes and motifs. Such a system can also help to immerse players by accurately fitting the mood of the music to the mood of the scenario.

3. Literature Search and Previous Work

3.1 A Generative, Adaptive Music System for MMO games

Hedges, Jim, Kurt Larson, and Chris Mayer. "A Generative, Adaptive Music System for MMO games." GDC Austin 2008. Sept. 2008. MP3 file.

This is Slipgate Ironworks' talk on "GAMS" or more recently "AGMS" – an ambient-based MMO solution that does not translate to FPSes. The system stands for "Generative, Adaptive Music System" and allows for the creation of zones. This talk provides the most inspiration for this particular inspiration, even though the developers specifically state that it works best for ambient music and "if you want to create "Ride of the Valkyries", this isn't the right tool to use. The solution for *Trigger Happy* suits a different need for the audio soundscape and will hopefully serve a different audience with the same goal.

3.2 Procedural Music in *Spore*

Jolly, Kent, and Aaron McLeran. "Procedural Music in Spore." GDC 2008. Feb. 2008. MP3 file.

In *Spore*, a system for procedural music is provided for the game's menus and creature creators. These areas are the ones that the player will spend the most time on, and therefore the ones most likely to become repetitive. The developers implemented a system that generates ambient melodies on the fly, partially using player input as a way to nudge the music in a particular direction. As in this system, the music is non-repeating, but the style of music does little to enhance mood/

3.3 *Dead Space* Sound Design: In Space No One Can Hear Interns Scream. They Are Dead.

Veca, Don. Interview. *Original Sound Version*. Web. 24 Feb. 2010. <http://www.originalsoundversion.com/?p=693>.

This interview discusses the role of music and sound design in *Dead Space*. Much like the *Left 4 Dead* solution provided below, the *Dead Space* developers went with a system for cueing audio tracks when necessary to add to the soundscape and create an appropriate feel of horror. They also discuss an audio scripting language used for the game, a set of 3d samples played to make the player feel fear, and "fear emitters" that act much like the audio emitter used in the *Singularity* audio engine. This is an opportunity to get an in-depth look into non-traditional use of composition.

3.4 *Left 4 Dead* Audio Commentary

Valve. Audio commentary for Left 4 Dead. In-Game Audio Commentary.

In their *Left 4 Dead* commentary, Valve employees discuss some elements of the audio of the game. One of the most interesting bits of commentary is about the adaptive audio in the game. In addition to the AI Director is the Audio Director, which allows the game to cue small pieces of music based upon the game state. While this is not as ambitious a system as proposed here, this is an excellent way to use cueing music based upon the world and adapting it to the game world. Transcript is available at http://left4dead.wikia.com/wiki/Developer_Commentary_(Left_4_Dead).

3.5 Computer models of musical creativity programs

Cope, David. *Computer models of musical creativity programs*. Web. 24 Feb. 2010. <http://artsites.ucsc.edu/faculty/cope/cmmc.html>.

On this section of his website, composer and programmer David Cope, the creator of Emily Howell, an AI composer, offers a selection of tools developed to create creative music through software. Cope helps to show how composition is simply mathematical rules – especially when listeners are very pleased with Emily Howell's music until informed that she is actually an artificial intelligence. While Cope does not work in games with adaptive music, some of his research is critical to software-driven audio creation and shows how far we have come in time.

4. Implementation and Deliverables

4.1 Plan of Action and Fallbacks

4.1.1 Attempt 1: "Slices" of Audio

The problem will be solved through the implementation of an adaptive music subsystem in the game. The *AdaptiveMusicManager* will be interface with the Singularity engine's *AudioManager* and tasking system. This *AdaptiveMusicManager* will be built on top of XACT for its implementation. It will have a significant number of individual "slices" of music, which are single-instrument measure-long clips designed to be ordered and played non-linearly. Content developers will need to think in a different fashion in order to develop these "slices", and the development of this audio will result in a set of "best practices" for future content and heuristic developers. In this type of system, content developers will often be required to develop the playback heuristics for the game in conjunction with the game developers and designers.

The adaptive music subsystem will accept in game data modifications through the use of an updater component, which will query the necessary *GameObjects* in the engine to collect a set of numbers. These numbers will each correspond to a variable defined in the Adaptive Audio Playback Tool, or AAPT. The AAPT allows content developers to nonlinearly compose their tracks by defining what variables the game world should be paying attention to (in correspondence with the developers) and what individual "slices" of music are allowed to play with others. The update component will run in a separate subtask in the Singularity engine and update once every few frames.

The *AdaptiveMusicManager* will then thread the various different tracks together based upon the data it is given. A major aspect of the implementation is the determination of what will and will not be "successful" for stitching together a track that is enjoyable and non-looping.

4.1.2 2nd attempt

The primary fallback for the system involves using a MIDI renderer to generate the music on the fly without pregenerated tracks. Because music composition is often mathematical, heuristics can still be used to piece music together. However, it will much more difficult to set the mood in this fashion. This system also will require more memory usage in order to generate the notes, and may potentially generate measure-by-measure rather than track-by-track.

4.1.3 Fallback

On the off-chance that both of these systems are unsuccessful, linear audio tracks will be delivered at the end of the content delivery phase. These tracks will be loops that replace the adaptive music.

4.2 Deliverables

There are three major deliverables for this project. Each is described in light detail below.

4.2.1 Adaptive Audio Playback Tool (AAPT)

The AAPT is a tool used for testing the adaptive audio slices and determining their properties and heuristics. Variables can be defined to output to XML, and will create a tag with the variable's name. The implementation in the *AdaptiveMusicManager* will have to recognize and read in these names. Some sort of content metadata will be associated with each of these variables. (For instance, if the "health" variable is within a certain range, the likelihood of playing a specific slice may increase.) The AAPT is crucial for defining the heuristics for playing musical slices, and these heuristics will be determined as needed.

The first version of the AAPT should be implemented and available around week 3 of development quarter.

4.2.2 *AdaptiveMusicManager*

The *AdaptiveMusicManager* is the subsystem within the game architecture that will identify the game state and determine the pieces of data used to determine which music slices should be played. It will load in an XML file created in the AAPT to determine what each heuristic modifies, and play back audio appropriately according to the defined ruleset.

The *AdaptiveMusicManager* should be implemented and integrated around week 5 of development quarter.

4.2.3 Content

The content to be delivered is made up of the musical "slices", which are short, measure-long instrument-specific pieces of music that can be sewn together with the system. Development of these pieces will be crucial, and the various instruments will need to work in harmony with each other. Various rhythms can be used, but some tracks will need to be marked not to play with one another.

These slices will be delivered within weeks 6-9 of development quarter, allowing time for adjustment and polish.

4.3. Requirements of Solution

4.3.1 Adaptive Audio Playback Tool (AAPT)

- The AAPT must output into XML or some other filetype readable by the Singularity engine and the *AdaptiveMusicManager*.
- The AAPT must allow variables to be changed by the content designer.
- The AAPT must be developed with C# and Windows Forms for convenience of the programmer and the content developer.
- The AAPT must allow content designers to select soundbanks/wavebanks and choose settings for tracks.
- A specific structure for file output must be developed for the AAPT.

4.3.2 *AdaptiveMusicManager*

- The *AdaptiveMusicManager* must accept world data from a subtask.
- The *AdaptiveMusicManager* must be able to read in data from XML or some other defined filetype.
- The *AdaptiveMusicManager* must determine which slices to play based upon the data given and an elimination algorithm.
- The *AdaptiveMusicManager* must have its content dynamically loaded – each piece has some sort of metadata associated from the AAPT.
- If the *AdaptiveMusicManager* has a choice of multiple slices to player, they should be randomly chosen based upon a weight provided by the data.
- Slices must be queued to play in sets of measures so the music does not sound incohesive. At the same time, the system should be able to prioritize new slices if the data changes quickly and significantly enough.
- The *AdaptiveMusicManager* should use XACT. As a fallback, Wwise and FMOD will also be considered if XACT proves impossible to work with.

4.3.3 Content Generation

- Slices must flow seamlessly into one another and combine well.
- Slices must be developed measure-by-measure, instrument-by-instrument.
- Slices should be developed in Apple Logic using MIDI for rhythmic consistency.
- Slices should be developed with particular rhythms in mind. These rhythms can and will help to define which instrument slices can layer on top of each other.
- Slices must be developed at a constant tempo and in a constant key. This can be revisited later if time allows.

4.4. Final Goals for Solution

There are a few overarching goals for this solution. First, the music system must use “slices” of audio in some fashion. This is significant, as it allows for more dynamic and mood-generating audio than an ambient solution. Next, the music system must be theoretically non-repeating for a significant length of time. This type of system cannot guarantee that the same selection of slices will be blended together at any given time, but the design of the system and the limitations of content development mean that it will at some point occur.

This music must also be appealing and mood-assisting. The development of the individual slices will allow the content creators to easily create moods and define rules to blend them together. Moreover, the inclusion and exclusion of individual slices in a particular variable allows a strong individual focus on mood-appealing elements. A “happy” set and an “angry” set could be defined that are exclusive, and a variable to switch or blend between them can be defined.

Finally, there should be a few separate non-development bits of information that comes out of this: a set of heuristics for *Trigger Happy* and an understanding of what types of information make good rulesets for the system. More importantly, an understanding of *why* these world is important and can help to field future work in the area. Finally, a set of best practices for non-linear audio content in such a system would be a fantastic thing to provide to those doing future work in the area.

Sela Davis
seladavis.com

(484) 919-4559 | sela.davis@gmail.com
273 Bennington Hills Court, West Henrietta, NY, 14586

Skills

Languages: C, C++, C#, Java, Bash, PHP, MySQL, Javascript, XML
Development Tools and APIs: DirectX10, Microsoft Visual Studio, Vi
Audio-Specific Tools and APIs: Audacity, Apple Logic, Ableton Live, Digidesign Pro Tools, Microsoft XACT
Operating Systems: Windows, Mac OSX, GNU/Linux (Debian, Gentoo, Red Hat)
Soft Skills: Interpersonal skills, teaching and training experience, team project experience, presentation development

Work Experience

Rochester Institute of Technology, Rochester, NY: Research Assistant (09/09-present)

- Designing and developing an architecture for benchmarking, recording metrics, and providing students a modular framework to assist game development in relevant classes.
- Framework will be used to develop *The Train Accident* in conjunction with the Games For Learning Institute at New York University.

Various Locations: Freelance Game Journalist: (05/09-present):

- Write articles, reviews, and editorials for various websites on an article by article basis.
- Attend and cover relevant conferences such as the Austin Game Developers Conference.
- Conduct interviews over telephone/Skype and in person.
- Featured at Play This Thing, Total Gaming Network, Nerd Girl Pinups, Elder Geek, and GamesPlusBlog.

SAP Americas, Rochester, NY: Knowledge Management Intern (05/07-03/08):

- Maintained the Field Services section of the SAP Corporate Portal and developed new pages.
- Worked with employees across various locations to acquire content and gather requirements.
- Developed standards for new Portal pages and developed and gave technical demonstrations to be used as future training materials.

Projects

Prometheic: Programmer, Designer (2009): 3D exploration game developed in DirectX 10.

- Developed one-sheet and co-designed mechanics, choices, and goals for the game.
- Refactored previous audio engine for others' ease of use and implemented 3D positional audio in-game.
- Built other components of 3D game engine such as screen management, 2D rendering, and user interface.
- Created all audio content.

Revenge of the Duzzles: Audio Programmer (2009): 2D shoot-em-up game developed in DirectX 10.

- Designed and built audio engine to integrate with XACT.
- Worked with other team members to implement some player components and various menu screens.
- Created all audio content.

Oh No! Banjo: Audio Lead. (2009): 2D rhythm game with a custom banjo controller.

- Arranged each track heard in-game by merging given banjo lines into original and licensed tracks.
- Made arrangements of some tracks to accommodate the banjo.
- Acted as sound designer and developed all sound effects.
- Maintained final control over all elements of the audio content development.

Professional Involvement

IGDA Rochester, Rochester NY: Co-Founder and Co-Coordinator (11/09-present):

- Acted as a driving force in the founding of IGDA Rochester.
- Contacted IGDA Board members in order to formalize the chapter and organize the local game development community within Rochester.
- Acting as co-coordinator and ran the meetings along with one other person.
- IGDA Rochester is a local chapter of the International Game Developers Association, a professional organization for game developers.

BarCampRochester, Rochester, NY: Organizer (01/07-present):

- Worked with a small team to organize and host BarCampRochester2, BarCampRochester3, BarCampRochester4, and upcoming BarCampRochester5 (April 2010).
- Solicited donations, reserved space, acquired meals, advertised to the public, and oversaw events.
- BarCamp is a free-to-attend "unconference" primarily rooted in technology at which all attendees are expected to present.

Education

Rochester Institute of Technology

Degree: M.S. Game Design & Development (09/08-present), GPA: 4.00
Major/Minor Tracks: Major track: Graphics & Engine Development; Minor track: Content Development

Degree: B.S. Information Technology (09/05-08/08), GPA: 3.53
Minors/Concentrations: Minor in Creative Writing; IT Concentrations: Digital Media, Digital Audio.

Dynamic Difficulty Adjustment and Storytelling in Multiplayer Competitive Environments

Capstone Research Topic by Chip Hilseberg

The Problem

There are two specific problems that I am attempting to tackle with this research project. They may at first seem to be unrelated, but they actually share a deep link that makes them well suited for a joint solution.

The first problem comes from the conflict between storytelling and interactive environments. On the one hand, designers want to communicate a powerful sense of world, character, and plot in their games. On the other hand, they want the player to feel in control of their actions and not sit idle waiting for cut scenes to finish. Gains in one area often come at a cost in the other area. Anyone who has played *Metal Gear Solid* can tell you that games can sometimes feel more like a movie when constructed in certain ways. The pace of a first person shooter (as *Trigger Happy* is) makes this problem even more difficult, amplifying any breaks in the action to the point where they quickly annoy the player.

This doesn't mean that story telling is impossible in an FPS however. *Half Life* and *Half Life 2* showed that in-game machinama can be used to tell a cut scene story in a less intrusive way. *Halo* and *Max Payne* featured solid enough stories to interest Hollywood movie makers. But all of these successes share one thing in common: their stories are told in a scripted, single player mode. What about multiplayer competitive modes? Can a designer convey the game's world without disrupting a Capture the Flag match? Can they do it with no scripted battles, no dialog, and little if any control at all? I wish to attack this open problem, and push past the half hearted attempts of games like *Unreal Tournament* with nothing more than weak back stories for the death match competitions.

The second problem is found in all games, but is particularly interesting in a multiplayer competitive context. It is known as dynamic difficulty adjustment or adaptive difficulty[1] ,and deals with the fact that players of infinitely different skill levels play the same game content. In single player games, the problem is often dealt with by offering a couple distinct difficulty levels that the player can choose from. However, some single player games go beyond that point by attempting to measure and dynamically adjust the challenges that a player faces.

A good example of this approach is *Max Payne*, which adjusts enemy HP and player aiming precision to keep the player constantly and precisely challenged based on their changing success rate[2]. Other solid examples include *Left 4 Dead,* which dynamically manages zombie spawn amounts and *Half Life 2,* which controls pickup quality based on player HP. However, not all approaches are successful.

1 http://www.gameontology.org/index.php/Dynamic_Difficulty_Adjustment

2 http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml

Racing games are notorious for the “rubber banding” algorithm which allows the AI cars to cheat and sometimes even teleport to the player's location in the name of creating challenge[3]. However, all it really does is anger the player who is not presented with a fair competition.

In multiplayer games the problem is even harder to solve because the designer has little control over the “difficulty” of the other players. There have been some systems implemented to approach this problem, but they each have their flaws. One approach is rating-based matchmaking, which tries to evaluate player skill in the form of a rating and tailor rewards to match the balance of ratings in a match. A game that does this is *World of Warcraft,* which rates its 2v2, 3v3 and 5v5 PvP “Arena” competitions based on a team's success against other rated teams. *WoW* attempts to ensure interesting competition by choosing similarly rated teams for battle, but if a match isn't available, it settles for not penalizing the worse team very much when they get slaughtered. This is ultimately fair, but it does nothing for ensuring a *fun* match between players of different skill levels.

Another approach is handicapping the better player or allowing the worse player to gain an advantage. An example of this is *Quake III,* which has an option in its multiplayer mode that allows players to assign a handicap to themselves. This handicap modifies the amount of damage received, allowing a weaker player to sustain more hits. The problem with this approach is that it can easily feel like the weaker player is cheating (because in a sense, they are). All it takes is one good player rating themselves low to throw off an entire match.

Finding a way to allow for balanced competition between players of many skill levels without crippling the better players’ experience or cheapening the weaker player's experience is an open problem.

The link between these two problem is that their solutions take the same form. Both of them seek to bend the multiplayer game experience in subtle ways to convey the designer’s will. For the first problem, the designer is conveying a certain style or theme, while in the second, they are conveying a certain difficulty level. Subtle methods of game play control could be equally applied to both problems.

Why is this Problem Important?

These problems are important for their own reasons. The first is important because modern multiplayer FPS games have very little of the emotional power of story driven games. They may be set in an environment from a well-crafted single player campaign, but rarely do they convey or even maintain the power of the context. I can set my avatar as Master Chief and play a map set in a Covenant Base, but in an online Capture the Flag match, all of the context drops away. The player’s actions are not shaped by the world design, only by the polygons that they collide with. Why can’t a player compete with other players while *also* taking something from the world and feeling a part of it? Some MMOs succeed in doing this, such as the rivalry between the Horde and the Alliance in World of Warcraft. Blizzard’s division of languages and races was an active design choice that contextualized and enhanced their PvP combat. Analogous tactics that mesh with the FPS game world could have similar benefits.

The second problem is important because multiplayer games suffer quite a lot from experience imbalances. Whether its new players getting stomped by veterans, random groups getting stomped by coordinated groups, or rating differences preventing friends from playing together, there are many opportunities for frustration and/or boredom. None of these situations are acceptable in single player games, revealing a distinct flaw in their multiplayer counterparts. An even partially successful solution to

3 Ibid.

this problem would do a lot to increase the number of fun, epic matches and reduce the number of ones that leave players feeling like they don't want to play another.

Goals

This research project has two main goals to be realized through one unified feature:

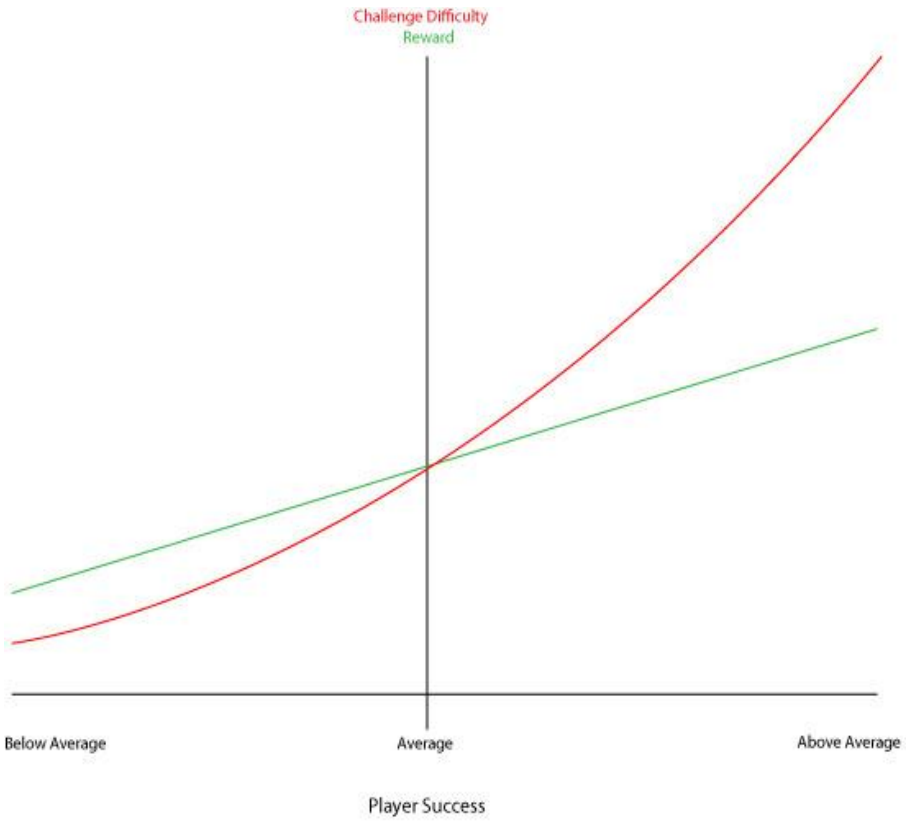
- Communicate the world of *Trigger Happy* in a way fitting for a multiplayer, competitive first person shooter. Specifically, communicate the style of the four organizations and of the combat entertainment industry.
- Create a mechanism for balancing player difficulty that meets the following criteria:
  - Allows players of different skills to play in the same game.
  - Encourages new players to improve, and assists them in the mean time.
  - Reduces the relative bonuses to better players in a way that doesn't cripple them.
  - Allows better players to show off in ways that compensate for this reduction.

The Approach

The mechanic that I'm proposing to explore for the this project is similar to the achievement system popular in many console games. In an achievement system, the designer sets out certain scenarios or challenges that are additional to, but synergistic with the existing game play. Examples can range from the banal ("Kill 50 people") to the odd ("Get a kill with every weapon in one match"). Once the achievement is gained, it is gained forever and usually displayed on the player's profile.

My mechanic would take the achievement concept and change it in three main ways. First, I would make them repeatable and only last for a single multiplayer match. The point of this change is to reward players that meet the challenge with a match specific benefit that helps them succeed. It isn't about nailing the achievement once so much as performing in a way that reliably collects them (and showing some style in doing it).

The second change would be to vary challenge difficulty based on how well the player is performing in the match. Existing achievement systems have challenges that vary in difficulty from ones that can be done without even knowing that they exist to ones that require careful setup. I wish to use this diversity to act as a difficulty adjustment technique, which gives player that are significantly outpacing the average exponentially harder challenges for linearly better rewards.



By using the challenges in this manner, I offer below average player's easier access to beneficial buffs without just giving them away. At the same time, better players have access to better buffs, but at exponentially higher difficulties. As a player trends toward above average, the reward to challenge ratio decreases more and more, meaning that they are given less and less assistance without giving them nothing to shoot for. At the same time, success at these difficult challenges is announced to the match players, compensating for the decreasing game benefits with "fame."

The third change would be to tie the challenges in to the story. Achievements generally come with some flare and style that gives them their own personality. For example, in *World of Warcraft*, there is an achievement for using the hug emote on every type of small animal in the game. I hope to use this characteristic to apply the "flavor" of the three organizations in the game (The ACF, The Society for Historical Beatdowns, and Impossible Possibilities) to the challenge requirements. For example, Impossible Possibilities has a mad scientist feel, so their associated challenges would focus on odd combinations and ways to kill people. I can also use it to flesh out some of the "entertainment industry" TV shows by sponsoring challenges for players to complete. For example, a historical reenactment show could sponsor "The Spartan – Use a wall to stop three incoming projectiles" as a sort of advertisement.

Research Literature

- http://www.gamasutra.com/blogs/EnriqueDryere/20091229/3959/Difficulty_Setting_in_Multiplayer_Games_Can_it_be_done.php
- http://www.gameontology.org/index.php/Dynamic_Difficulty_Adjustment
- http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml
- https://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-019.pdf
- http://portal.acm.org/citation.cfm?id=1178573
- http://www.gamasutra.com/view/news/18376/Feature_The_Designers_Notebook_Difficulty_Modes_and_Dynamic_Difficulty_Adjustment.php

Deliverables and Integration

- A game mechanic or set of game mechanics that achieve the previous goals.
- A user evaluation of the game to determine if it achieves those goals. This will be done by play testing the game with a sample of users and recording their answers to questions about balance and style.

When is it Complete?

This research will be complete when:

- A game feature has been created.
- It has been integrated in the game.
- It has been play tested by non-members of the team and evaluated.
- It has been improved based on that feedback (time allowing).

Design and Implementation of a Componentized Multi-Core Game Engine

Capstone Research Topic by Jonathan Lobaugh

The Problem

The specific problem I am addressing in this research project is the integration of multi-core processor architecture and game engine design. I hope to integrate two different technologies to build an engine that is both easy to use and optimized for multi-core processors.

The key development construct for most game engines is the game loop. The game loop construct has been at the core of most of the commercial engines and has worked well for single processor systems. However, with the advent of multi-core processors, game engines have been unable to adapt the game loop to fully utilize multi-core processors. With the current market share of multi-core processors reaching over 50%, the need for game engines to adapt and embrace the multi-core processor is becoming a huge motivation for a shift in game architecture.

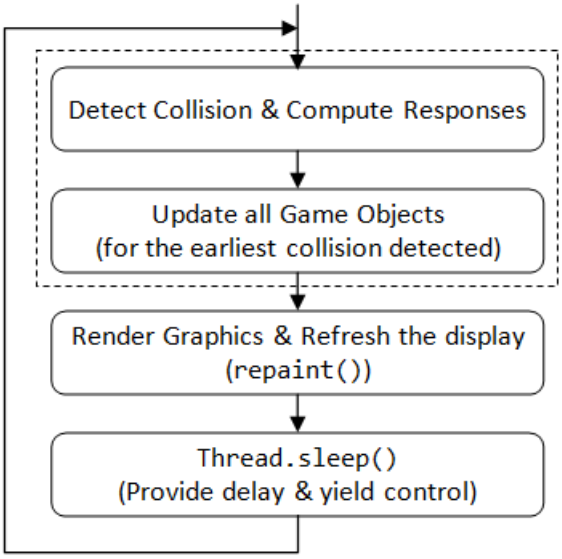


Figure 79 : Game Loop Control Diagram

Since multi-core processors were introduced developers have been attempting to move the game loop from a single-core process to a multi-core process. The shift produced engine architectures that were built around forms of data synchronization constructs such as critical sections, reader/writer locks, or barriers and context switching mechanisms. Data synchronization constructs (aka locking constructs) provide a means for developers to re-synchronize the data across the multiple cores; without such synchronization, data would become unreliable. Context switching mechanisms are a means by which multiple threads are able to run on a single processor.

However, as a cost to the synchronization and thread management, both locking constructs and context switching take valuable time and memory to execute. Many times the locking constructs can cause large slots of execution time to be lost. Each time a thread must synchronize, the thread must stop its execution which causes a context switch. These two actions are a huge loss of processing time when dealing with game engines.

The job of an engine designer is to understand these locking issues and design a means for developers to minimize their use. However, many times it is difficult to minimize locking constructs without building the engine specific to the development of the game. To further complicate the system, many times design of the game makes it impossible to eliminate the locks from the engine.

To take full use of the multi-core processors, the time used to synchronize data and switch threads among cores needs to be minimized, or eliminated if possible. Intel's Threaded Building Blocks (TBB) library attempts to minimize the time lost to thread management by eliminating the need to switch threads. By utilizing small tasks the library is able to minimize the amount of time lost to context switching. Data synchronization is still an underlying issue, one that the TBB library does not fully address.

To fully utilize multi-core processors and architecture needs to be designed that minimizes the usage of context switches as well as forms of data synchronization locks. If an engine architecture is able to minimize these two slowdowns while maintaining an easy to use system, then game engines will finally move into the realm of multi-core applications.

Why is this Problem Important?

As game development as matured over the years, the need for more processing time has become a huge limiter of what features have been implemented in games. Processes such as Artificial intelligence and physics have been limited in order to manage the frame rate of games. The hardware industry has answered back with multi-core processors, expanding the amount of computational time available to developers. However, the game loop was never intended to exist as a threaded process and most implementation that attempt to amend the game loop end up with very little benefit compared to the amount of overhead.

In order to embrace the technological improvements of multi-core processors, engine design is going to have to explore another architectural model. Models that are able to abstract the process of multi-core development while minimizing the overhead of the multi-core development. When engines adopt this approach, then more process time can be opened for extended running processes.

Goals

This research project has two main goals:

- Create an engine framework that can integrate a task-based threading model and component-based entity architecture.
- Create a mechanism for allowing threaded rendering with a constant time optimization of draw calls.

The Approach

The architecture I am proposing is a combination of techniques. As describe above, the issue with game engine design and its subsequent move towards multi-core processors has been the steadfast use of the single threaded game loop. Techniques and libraries have been emerging on the forefront that has taken the threading models from the single processor to a higher abstraction in the form of tasks. The core utilization is quite high and allows for quick unitized tasks to be completed in parallel. Intel's Threaded Building Blocks have built this model into the library. The technique make use of every core of the processor while minimizing the context switching that is usually necessary when running threaded systems.

This threading approach integrated with a component-based entity architecture will be the direction I feel will provide the most processor utilization. Where as the tasking system allows functional separation, the component-based entity architecture allows data separation. When combined properly the two models allow threading with both functional and data separation, completely individuating each task. What this means for the engine, is a threading model with little to no data synchronization locks. The ability for an engine to manage and execute game logic across multiple threads with no data synchronization or context locks is a means of utilizing multi-core processors to their fullest.

Research Literature

- "Designing the Framework of a Parallel Game Engine - Intel." Intel Software Network communities - Intel. Web. 24 Feb. 2010. <http://software.intel.com/en-us/articles/designing-the-framework-of-a-parallel-game-engine/>.
- Gabb, Henry, and Adam Lake. "Gamasutra - Feature - "Threading 3D Game Engine Basics"" Gamasutra - The Art & Business of Making Games. 17 Nov. 2005. Web. 23 Feb. 2010. <http://www.gamasutra.com/features/20051117/gabb_01.shtml>.
- Gasior, Geoff. "Valve's Source engine goes multi-core - The Tech Report - Page 1." The Tech Report - PC Hardware Explored. 13 Nov. 2006. Web. 24 Feb. 2010. <http://techreport.com/articles.x/11237>.
- Lorenzon, Jorge A., and Esteban Walter Gonzalez Clua. "A Novel Multithreaded Rendering System based on a Deferred Approach." SBGames 09. Print.
- Mönkkönen, Ville. "Gamasutra - Feature - "Multithreaded Game Engine Architectures"" Gamasutra - The Art & Business of Making Games. Gamasutra, 06 Sept. 2006. Web. 24 Feb. 2010. <http://www.gamasutra.com/features/20060906/monkkonen_01.shtml>.
- Slade, Matthew, Ken MacGregor, and Edwin Blake. The Development of a Multi-threaded Game Engine. The Development of a Multi-threaded Game Engine. 2008. Web. 23 Feb. 2010. <http://shenzi.cs.uct.ac.za/~honsproj/cgi-bin/view/2008/packham_slade.zip/packham_slade/downloads/report_matthew.pdf>.
- Tagliasacchi, Andrea, Ryan Dickie, Alex Couture-Beil, Micah J. Best, Alexandra Fedorova, and Andrew Brownsword. "Cascade: A Parallel Programming Framework for Video Game Engines." Print.
- Zamith, Marcelo, Mark Joselli, Luis Valente, Esteban Clua, Anselmo Montenegro, Regina Celia P. Leal-Toledo, and Bruno Feijo. "A Game Loop Architecture with Automatic Distribution of Tasks and Load Balancing between Processors." Brazilian Symposium on Games and Digital Entertainment 8 (2009): 5-9. Print.

Deliverables and Integration

- The game engine framework source code
- The game engine framework one-sheet
- Small Demo of the Game engine
- Performance measurements and comparison data.

When is it Complete?

This research will be complete when:

- A game engine framework has been developed.
- It has been integrated into the game.
- The performance has been measured against other multi-core solutions.

Variable Style Deferred Rendering

Capstone Research Topic by Eric Moreau

The Problem

The topic of my research for our capstone project is the ability to create multiple art styles through a deferred renderer. The reason I am pursuing this topic is because of the art style that *Trigger Happy* utilizes multiple art styles to set up its environment. The characters and the environment each have a distinct look to them that is beyond the reach of a standard deferred renderer. A standard deferred renderer use a two-pass system. The first pass takes a snapshot of what is being rendered on the screen and applies shading model effects. This snapshot is often known as the *G-Buffer*. The *G-Buffer* holds all the data that is going to be sent through the renderer. The results from these effects are then output to render targets and fed into the second pass. The second pass renders out the final scene using the textures created from the render targets in the first pass. Since the whole scene is looped into one lighting model post-process, it makes it hard for multiple art styles.

Common deferred renderers render out to at least three render targets in order to achieve basic results. These render targets create textures for original pixel color, normal maps, and specular maps. Other maps have been known to implement a second pass before the final render pass to achieve higher detail in their scenes. The Leadwerks Engine utilizes a second pass that includes a diffuse lighting and specular reflection render targets added to the scene.

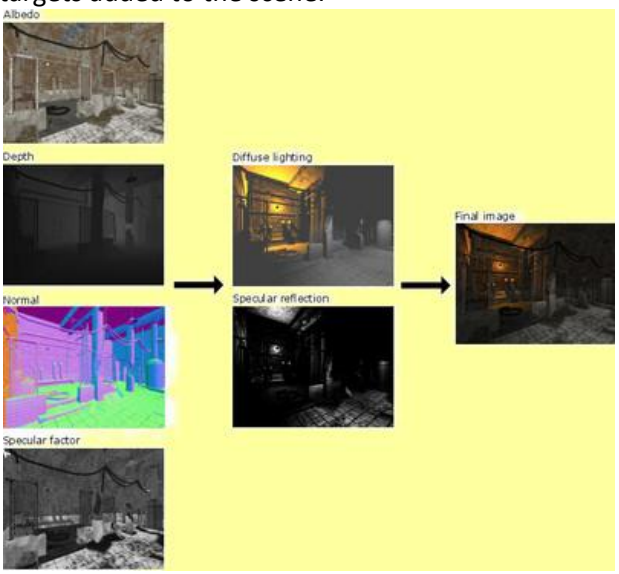


Figure 80: Deferred renderer implemented in the Leadworks Engine 2.0

Why is this Problem Important?

Handling multiple art styles is usually handled using a forward rendering process as it can handle more object specific post-process techniques. Deferred Rendering excels in performance over forward rendering in the sense that it is only doing one draw call per frame. Forward rendering takes multiple

draw calls per frame. Because of this single draw call, deferred rendering also scales nicely when more lights are added to the system. This scaling factor comes into play most notably in enclosed environments where more lights are required to light the area. Deferred rendering does not work so well in outdoor environments since it generally setup by one large point light that represents the sun.

There has been a long debate as to which style of rendering is better. As of right now there is no definite answer since each system has its own strengths and weaknesses. The correct answer at this time is it is really system and game dependent. There are limitations to deferred rendering outside of the post-process limitations. Deferred rendering does not handle transparencies in scenes that well. This is due to snapshot approach to the scene. From there it is hard to determine draw order of objects in the scene. Another issue is hardware anti-aliasing in deferred rendering. Many hardware architectures are incapable of applying standard hardware MSAA anti-aliasing to scenes. There are ways around the anti-aliasing issue but engines are forced to take performance hits in order to achieve the upscaling and downsampling required to perform the techniques.

While the exploration of the transparency and anti-aliasing is outside of the scope of my research, they are both worth noting as limitations of the deferred rendering pipeline. On the highest concept level, adding the ability to vary art styles inside a deferred renderer moves away from the standard deferred rendering pipeline model and more into a forward renderer model, though not entirely.

The Approach

My development plan can be broken down into two steps. These steps are to ensure that if there is a point in my research where I feel that my desired results cannot be achieved, I can assess the situation early and often without delaying development or break already existing portions of our engine development.

Implement Basic Deferred Renderer

The first step to achieving my end-goal is to implement a deferred renderer. Without a basic renderer to draw from, it will be difficult to understand where and how to break up the G-Buffer. Already implemented in the Singularity Engine that *Trigger Happy* is to be running on is a forward renderer system. When the deferred renderer is complete I will do a performance assessment of both rendering systems. Comparing the performance difference between the two system will give me an understanding of which system is faster for the Singularity Engine as well as give me an indication of how much of a performance hit a final solution would take from the engine.

Compartmentalize the Renderer

The second and more challenging of the two steps will be the compartmentalizing functionality of the renderer. Users working with the system will need to be able to identify what types of art style they want their objects to render as. From there each style will need to identify what textures they will need to achieve the desired effect. I will be taking a page from the Leadwerks Engine approach by using a second pass to render textures that apply to all styles; such as specular reflection and diffuse lighting. Below is a flowchart for the system in its final stage.

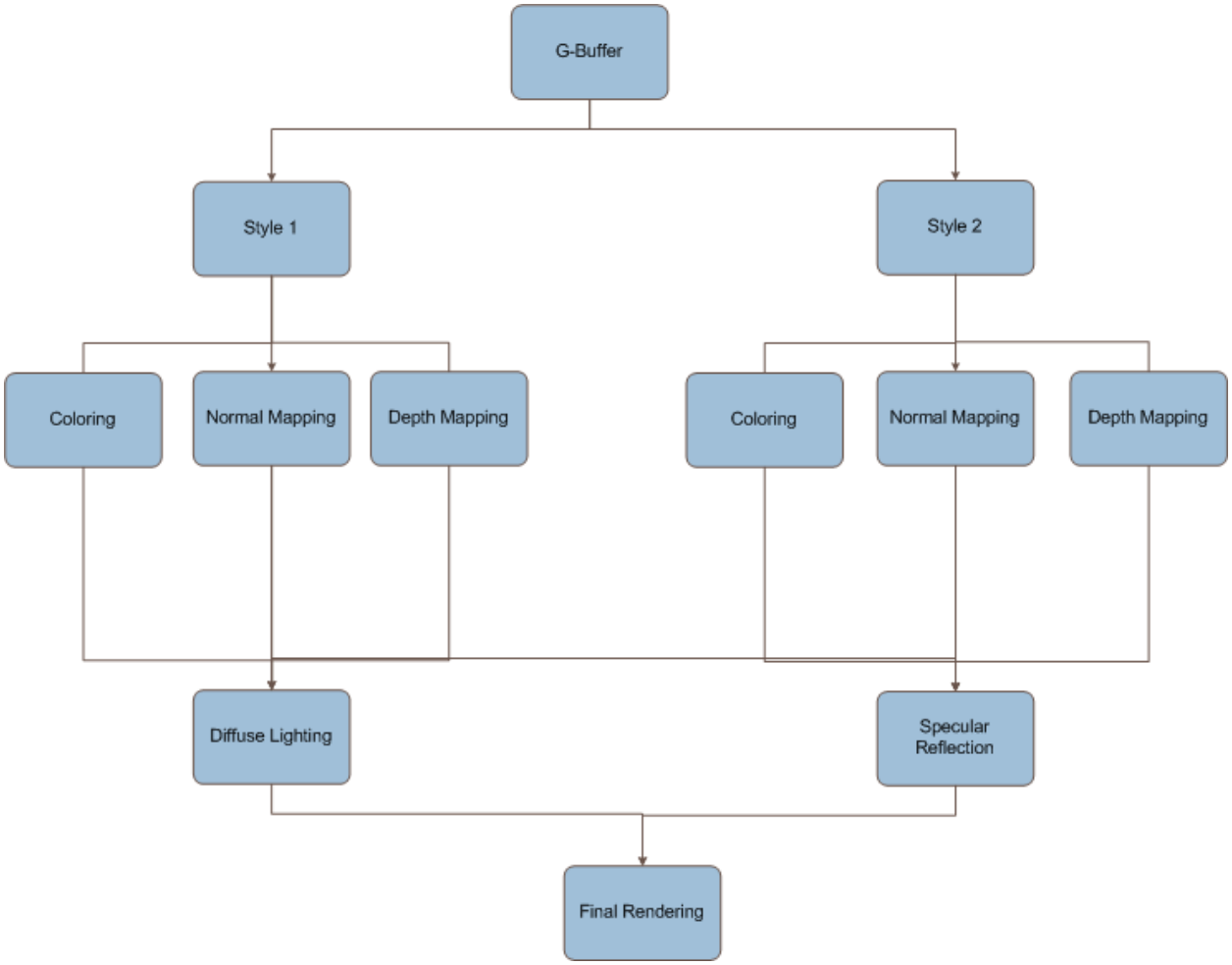


Figure 81: Deferred Rendering flowchart proposition

Goals

- Implement a deferred rendering system
- Be able to compartmentalize objects in the scene to apply separate post-process techniques.
- Be able to apply separate post-process techniques to separate compartments.
- Bring all compartments back together for a second pass

Deliverables and Integration

The deliverables for this system will be a fully integratable rendering system that allows for varied post-process techniques using a deferred rendering core. Integration with the game and the engine will be fairly simple due to the engine's component system. The component system allows for base game objects to add modular components to achieve various levels of functionality. The game objects this research will focus on objects drawn on the screen that require a rendering component. As of the time

of the writing of this proposal, the Singularity Engine is capable of handling the forward rendering system and is capable of sustaining a deferred rendering component to replace the forward renderer.

One of the larger differences in the system will be how effects are handled in relation to the deferred system. In order for the deferred rendering system to work, objects will need to render to various render targets that will create the desired look and feel for *Trigger Happy*.

The fallback plan for this system will be to use the forward rendering system that already exists in our engine.

When is it Complete?

- Deferred Rendering framework has been developed
- The rendering system has been fully integrated into the Singularity Engine
- The rendering system is capable of compartmentalizing objects into style groups to render different post-process effects
- The compartments can be combined into one scene to render common post-process maps and rendered to a final scene
- Performance has been measured against the forward rendering system already in place in the Singularity Engine

Research literature

Much of my research has come in the form of community discussion on various websites with various papers on Deferred Rendering thrown into the mix. These discussions offer reasonable warnings and point / counter point comments on the deferred rendering process. I will use these discussions, papers, demonstrations, and any other future findings towards my research

Resources:

Deferred Particle Shading - http://download.unity3d.com/blogs/nf/files/page0_blog_entry73_1.pdf

Deferred Rendering in Killzone 2 - http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

Deferred Shading Implementation - http://www.gamedev.net/community/forums/topic.asp?topic_id=463293

[C#] [HLSL] Mixing deferred rendering and forward rendering - http://www.gamedev.net/community/forums/topic.asp?topic_id=559161

Lighting in Deferred Shading - http://www.gamedev.net/community/forums/topic.asp?topic_id=482654

Forward vs. Deferred Rendering – http://www.gamedev.net/community/forums/topic.asp?topic_id=424979

Michael Deering – Proposed Deferred Rendering Technique in 1988 http://en.wikipedia.org/wiki/Michael_Deering

Deferred Shading Shines. Deferred Lighting? Not so much
http://gameangst.com/?p=141

GPU Gems 2: Chapter 9 – Deferred Shading in S.T.A.L.K.E.R.
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html

Lighting Rendering Architecture Doubt – Discussion on multipass and singlepass lighting architectures -
http://www.gamedev.net/community/forums/topic.asp?topic_id=424468

Deferred Rendering in Leadworks Engine -
http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf

Deferred Rendering Sample -
http://www.codesampler.com/usersrc/usersrc_7.htm

Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects -
http://graphics.cs.uiuc.edu/~kircher/inferred/inferred_lighting_paper.pdf

http://lightindexed-deferredrender.googlecode.com/files/LightIndexedDeferredLighting1.1.pdf

Deferred Rendering Implementation Example -
http://www.catalinzima.com/?page_id=41

Game Creator's Newsletter – Issue 66 July 2008
http://gdk.thegamecreators.com/data/newsletter/newsletter_issue_66.html
Deferred Rendering in Frameranger
http://directtovideo.wordpress.com/2009/11/13/deferred-rendering-in-frameranger/

Maya-based Creation and Assignment of Component Attributes for a Component-based Game Entity Architecture

Capstone Research Proposal – Nicholas Wilsey

Background

Traditionally, game programmers have represented game entities through an object-oriented, hierarchical tree-based structure. This structure has been used to produce generations of games, but by using it, programmers have always been required to deal with its inherent issues. Most obvious of those issues is the fact that often times, programmers cannot describe every set of game entity relationships through a directed acyclic graph. And when they try to add functionality to the nodes (entities) in such a situation, they end up having to reorganize the graph each time they do it. That reorganization, fortunately, is relatively easy. Maintaining an acyclic graph after doing so, however, is generally not.

To deal with such a hurdle, programmers often try to avoid it all together by moving data and functionality up and down the graph (rather than reorganizing it). Sometimes these moves are sound, but often times they go against the key object-oriented principles that they hoped to follow when they chose a hierarchical structure. For instance, when one moves data up the graph, the "highest" nodes often begin to misrepresent what they truly are and become very bloated. When data is moved down the graph, it is often done by duplicating code, and sprinkling it around to the classes that need it. In the end, all functionality has its place, but given the data structure, one would not always find it where s/he might logically expect it to be.

To avoid these sorts of issues (and there are others), some game programmers have stopped using a purely inheritance-based architecture to represent their game entities. Instead, they have begun taking a composition-based approach. Programmers often refer to this method as a component-based architecture, and it represents a fundamental shift in game entity representation.

Component-based models flatten the typical hierarchical structure, almost making the concept of child/parent nodes and what that represents (in terms of function) irrelevant. In this approach, instead of adding functionality to an object through a long hierarchy, programmers compose objects of “behaviors” and “attributes”. Behaviors define functionality, and attributes act as the inputs and outputs to those behaviors. There are a few more details involved, but ignoring those for now, behaviors and attributes combine to compose “components”. These components are what we attach to individual game entities, and they are, essentially, what define a particular game entity. For the development of *Trigger Happy*, we have adopted such an approach.

Problem

To use a component-based system, there must be a way to define component attributes and behaviors, assign them to unique game entities, and then edit their individual values. This method should be simple to follow and easily achievable by any member of the team, regardless of which team one is on (i.e., the asset creation, engine development, or gameplay programming team). Moreover, when the game engine reads in and composes game entities at runtime, that process should be relatively fast and require as little overhead as possible. If the engine reads entities from a file, for instance, that file should contain the entity definition as well as assigned attributes and behaviors; reading any of that data from additional files would be considered unneeded overhead.

Defining a list of editable component attributes and behaviors is rather simple – one could do it in a simple text file. And one would not generally consider assigning components to unique game entities much harder either, especially if s/he uses an XML-type file schema and is able to edit those files. However, a text-based editing system is not necessarily the most intuitive way to go about things. Programmers may easily parse XML, but it is a cumbersome task to do so. Moreover, the mere fact of looking at code (or anything that resembles code) may simple be too daunting a task for an artist to easily handle. We need a system that can work for everyone, and is not needlessly cumbrous from tedious text file editing. There is no readily available system for us to use, so we will have to implement our own.

Claim

We already plan to use Autodesk's Maya software to define our game environment and the world position of its entities. I think that we could use Maya to help assign our components as well. The Maya API completely unlocks all of its features to C++ and Python programmers, giving them low-level access to all data required to make plug-ins. Wielding that power, I think that I could write a plug-in that would meet all of our needs.

Goal

My goal is to build a Maya plug-in that will allow us to easily and quickly define, assign, and change component attributes for our component-based game architecture. As stated previously, this system should give users the ability to easily define and edit component attributes, and then assign them to individual game entities. Moreover, it should help make the loading and construction of entities at runtime as fast and simple as possible. Finally, it should do these things in an intuitive way, with a very flat learning curve.

Method:

There are two ways to extend Maya functionality. The first is Maya's C++ API. This API provides access to every element within Maya, including all scene data, dependency graph information, shaders, file translators, etc. The second way is with MEL (or Python) script. MEL is the internal Maya scripting language, and is typically used to design user interface elements. One may also embed it within C++

code, and it will execute within Maya as if one had directly input its commands into Maya's internal command shell.

Using these two features, I would design and implement a C++-based plug-in that would be loaded and run within Maya. MEL script would allow me to create a user interface that is not only fully integrated into Maya, but also intuitive to use and easily extendable. In addition, I would use the C++ API to build the entity-defining components we would like to have (granting easy access to their behaviors and attributes in the process), and a simple way to assign them to any node within a scene.

With access to the dependency graph, particular components could be dragged-and-dropped directly onto the objects they would compose. Likewise, with a custom made GUI, editing those component attributes values would be as easy as changing a few values within a GUI field element. I would seamlessly integrate all component information into the data that Maya already assigns to its meshes, and export it all from a file type that we would easily parse with our game engine.

Conclusion

Maya will help give us the power to easily make a component-based game-entity-architecture a reality. Instead of designing and building an interface from scratch – even trying to use what could be a very tedious and cumbersome XML-based file approach – we could use everything what Maya already provides. Instead of editing text files, we could simply drag and drop components (thereby assigning them) directly onto individual objects within the scene. Those connections would be visually apparent to everyone, and easily changed through our custom-made GUI elements. Moreover, with full access to all data and file translators, we could easily define a custom file layout and optimize it for our game loading needs.

Switching to a component-based architecture, especially from a hierarchical approach, is not necessarily a simple task. If we can do anything to make our development process easier and ourselves more productive, we all think we might as well do it. And by using the power of Maya as we have outlined here, I think that we can certainly do that.

Research Literature

Autodesk Maya Online Help. Autodesk. Web. 22 Feb. 2010. <http://download.autodesk.com/us/maya/2009help/index.html>.

Complete Maya Programming. 1st ed. Vol. I. Morgan Kaufmann, 2003. Print

Complete Maya Programming. 1st ed. Vol. II. Morgan Kaufmann, 2005. Print

Et cetera