2015-04-28

Xeero: A 3D Action-Puzzle-Platforming Game

Daniel Acito
Worcester Polytechnic Institute

Xeero: A 3D Action-Puzzle-Platforming Game

by

Eric Anumba, Daniel Acito and Anthony Sessa

A Project Report

submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Interactive Media and Game Development

29 April 2015

Approved

Brian Moriarty, Project Chair

Britton Snyder

Keith Zizza

Abstract

This report discusses the design and development of *Xeero*, a 3D action-puzzle-platforming game constructed from our own custom engine, original art and sound assets. Despite a small development team, we strove to create a highly-polished and marketable interactive experience. We explain the methodology employed, results gained, and challenges faced by each member of the team in pursuit of this goal.

Acknowledgements

We owe significant thanks to a myriad of contributors who, without their support, the completion of this project would not be possible. Thanks to our thesis advisor Brian Moriarty, who provided us with much invaluable insight with his experience in the video game industry, and encouraged us throughout the entire development process.

Thanks to Britt Snyder and Keith Zizza, who served on our thesis committee. Additionally, Zizza was a valuable resource regarding audio development; we are especially thankful for him loaning Dan Acito some of the hardware he needed for creating the music, as well as providing guidance throughout the entire music developing process.

Our friends Caitlin Malone and Dan Manzo helped us design and implement a testing protocol which we were able to use to gain much-needed user feedback. These tests were extremely helpful and could not have been accomplished as easily without their assistance. Caitlin went on further to assist us in writing a business plan that our team plans to expand on in the near future.

We’d like to extend our thanks to the WPI IMGD community. Our fellow students took time to test our game, and provided us much encouragement. Their excitement to see us succeed was a driving factor for us.

A special thanks to Emily Hensley, who is a constant source of inspiration and whose faith and encouragement spurred Eric Anumba on through the course of the project.

Finally, thanks to friends and family. At no time were we ever short of support, and we are very appreciative of all of you.

Table of Contents

1. Introduction

This paper will discuss the IMGD Master's Project *Xeero*, a 3D action-puzzle-platforming game developed for PC. The team consists of Eric Anumba, Daniel Acito, and Anthony Sessa. Eric acted as the technical developer, designer, and visual artist. He produced the game binaries, designed the mechanics and levels, and created the 2D and 3D art assets. Dan was in charge of audio development; he designed the sound effects to be utilized throughout gameplay, as well as the music. Anthony acted as the team's producer; he designed production plans and schedules to ensure that team was making adequate progress every week.

Our motivation for this project was to design and implement a game that would be marketable; it is our intention to continue development after the completion of this project, and have *Xeero* commercially published.

Xeero will be released in stages, so that additional content may be distributed to players over time**.** Releases will be episodic; our initial release will contain only a few levels, but each successive installment will add more levels, mechanics, and story beats. Between updates, players will have access to a level editor that allow them to create and share levels using the mechanics that have already been released.

Over the course of the project, the team has been exhibiting and testing the game. *Xeero* has been demoed at PAX East and other local exhibitions, and has been playtested by dozens of students and players.

2. Background

2.1. Concept

"Is your computer running slowly? Suffering from data corruption and software glitches? Your machine may already be infected by The Alpha Virus! Before you throw your computer away, try Xeero! This program will run, jump, and Digitize its way through your system, stomping out malware and bugs! Be a hero, buy Xeero!"

The game is centered on the titular character, Xeero, an anti-virus program navigating a computer system devastated by the Alpha Virus, fighting and defeating malware and (literally) stomping bugs.

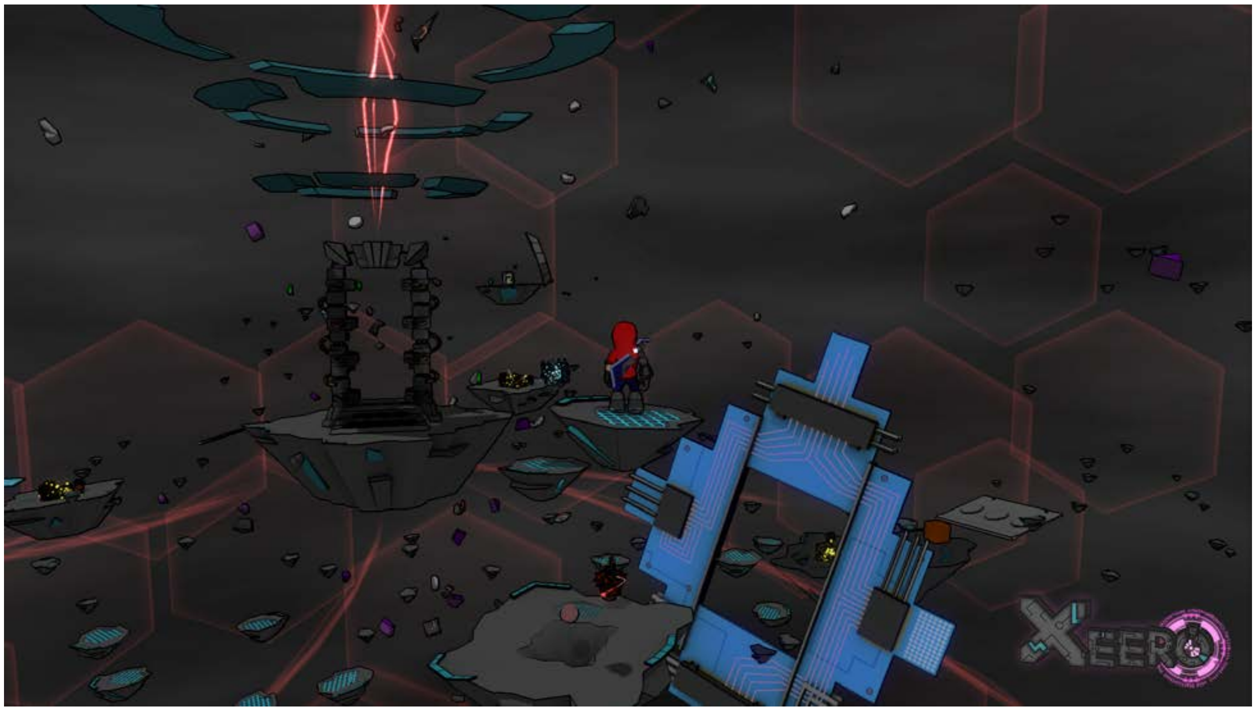


Figure 1: The world of Xeero

The world Xeero inhabits is treated as a single, infected computer, composed of several separate infected "programs," represented by the levels the player experiences. Each program contains at least one bug for the player to find and eliminate. Doing so fixes "corruption" in other parts of the system, allowing Xeero to progress to later levels. However, to find the bug, Xeero must navigate through the waves of malware infecting the programs and the hazards of the system's security (e.g. firewalls and gates).

Xeero has one essential skill: the ability to *digitize* certain components of the world, giving him new functionality and abilities. He can digitize blocks to allow him to reach high or far away areas, bombs to destroy obstacles, and upgrade chips to give him new capabilities.

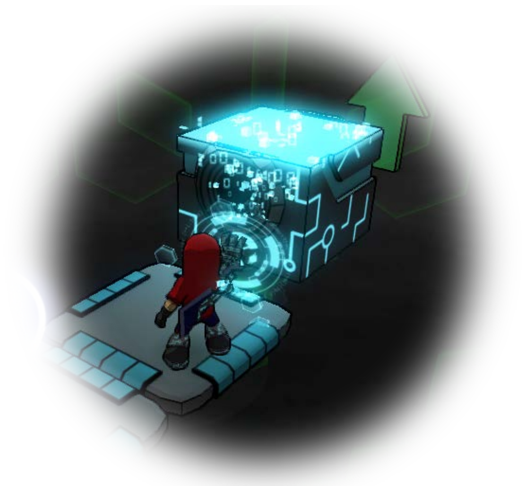


Figure 2: Xeero digitizing an object

One of the first abilities Xeero acquires is the control over Reaper, a telekinetic blade. This tool allows him to destroy malware and free their corrupted data. The player is given offensive and defensive abilities with Reaper (like a basic attack and the ability to block), and lets the player engage in combat with the enemies that appear in the game.

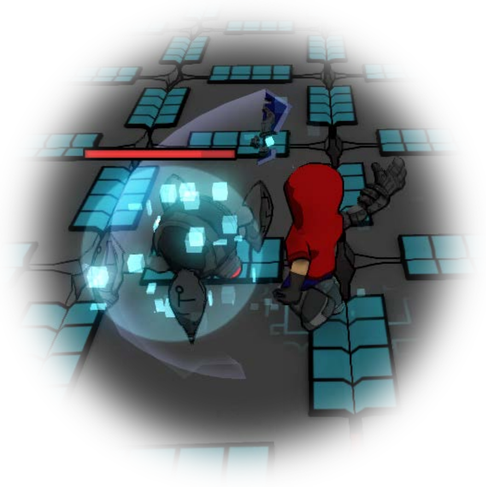


Figure 3: Xeero attacking with Reaper

2.2. Experience Goal and Audience

The game was designed to be played by established fans of 3D action games and platformers, such as those available for popular consoles like the Microsoft Xbox and Sony PlayStation. Action and platform games focus on the player having quick reflexes and the ability to perform controller moves with high precision. These players derive satisfaction from completing challenging levels and honing their skills. Thus, the game seeks to evoke satisfaction from the player, while also offering tough but forgiving gameplay.

The game seeks to evoke a sense of satisfaction in the player with themselves, by doing or witnessing a moment in gameplay that makes them feel like they have impressively mastered the rules of the world. To that end, the *Xeero* is focused on having the user experience two types of “moments” when playing the game.

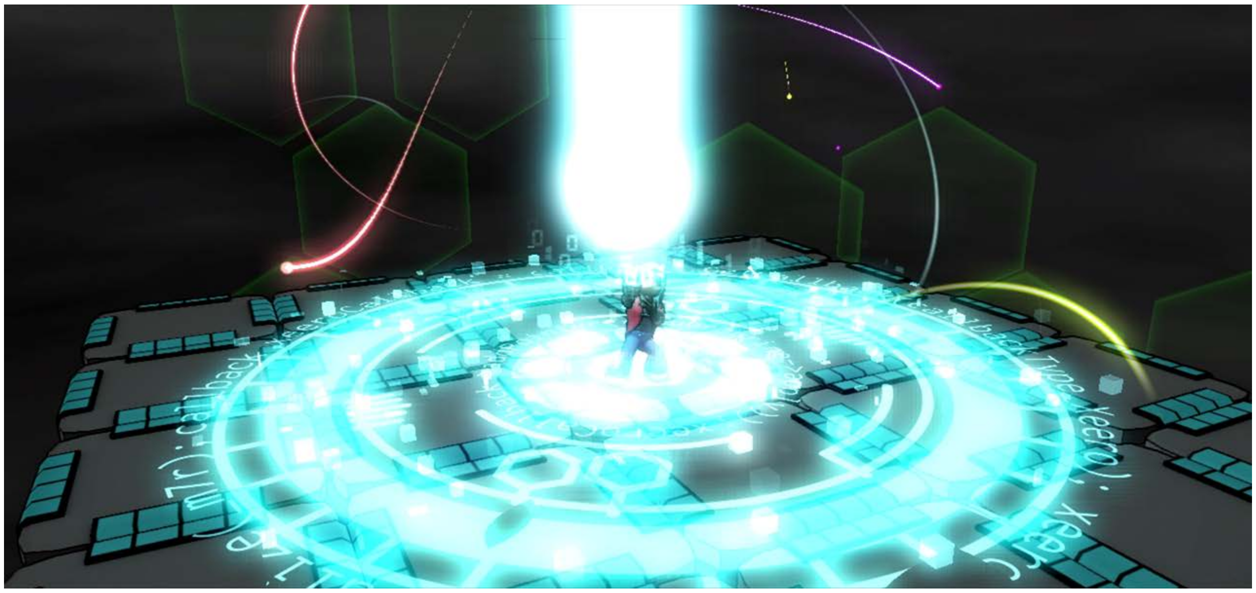


Figure 4: Xeero acquiring a new ability

Unlocking an exciting new ability. The player is given “upgrade chips” over the course of the game, each giving the player a new ability. At the moment of acquisition, a burst of light and energy (see the figure above), emits from the player, to reward the player for their gain.

Additionally, the new ability should be exciting and interesting to use, and at first use, the player should feel immediate satisfaction.

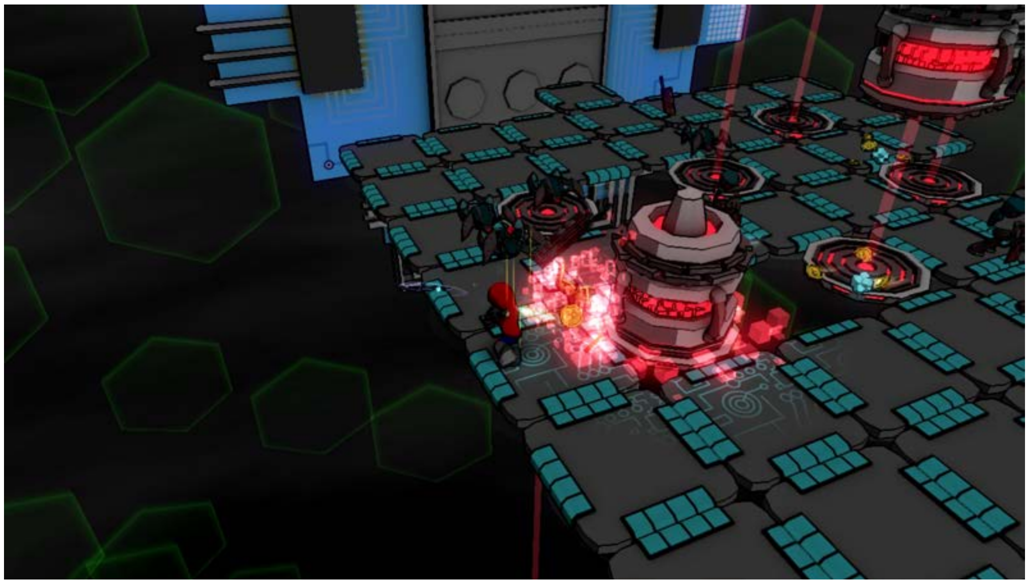


Figure 5: Combat in Xeero, aided by compressors

Combined interaction of game mechanics. The player is given tools over the course of the game and sees a variety of objects, each with their own effects or dangers. In certain moments, several different mechanics should interplay in a way that works to the player's advantage (either by helping them swiftly and deftly win a fight or clear a puzzle), and the player should feel a sense of satisfaction when the gameplay just "clicks."

In the other moments of gameplay, the player should be constantly engaged and challenged. Each puzzle or encounter of the game should be challenging for the player to complete, requiring heightened reflexes and the player to exercise practiced mechanics, but also should be flexible in allowing the player to fail early and often. The player is allowed to "die" in the game, either by falling off of the map into the program-wide firewall below, or by running out of health by getting hit by enemies. On death, the game should quickly reset and drop the player off right where they failed so they may attempt the section of level again. Failing at a puzzle should be of little consequence to the player, and they should be allowed to quickly try again until they execute the section correctly. Thus, the game should offer ample checkpoints and very quick respawn animations.

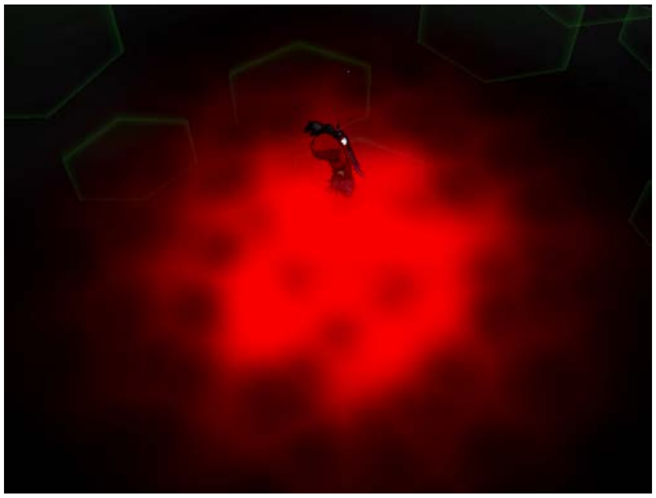


Figure 6: The program-wide firewall threatening to destroy Xeero

2.3. Concept Origin

2.3.1. Origin at UK and Entering WPI

Eric originally created the design of the game as a class project in his undergraduate computer science program at the University of Kentucky. At the time, in second half of his senior year, he attended the program's first game programming course, and began crafting *Xeero*, then called "Debugger," as the final project for the course.

The course was focused on using the Microsoft XNA Framework, a platform with which Eric was familiar. Since his freshman year, he had been working with the framework to develop games in his spare time. By building a series of games as side projects, he sought to familiarize himself with as much of the game development process as he could. He studied game programming, focusing on C# and XNA, covering graphics, physics, multi-threading, database, and network programming. He leveraged his interest in drawing and character creation to start learning how to create and animate 3D models using Blender, a free 3D asset creation program, and to draw digitally with Adobe Photoshop and Illustrator.

Eric had built a series of other games over the course of his undergraduate studies. His first large project, an ambitious online, multiplayer collectible card game, allowed him to study networking and database programming and learn the foundations of using Blender and Photoshop to create assets. He tried his hand at building a game engine from the ground up using OpenGL, and he worked with a team of other students to develop an arcade-style game, “Boom! Zap! Pow!” for the Microsoft Kinect for Windows (pictured below).

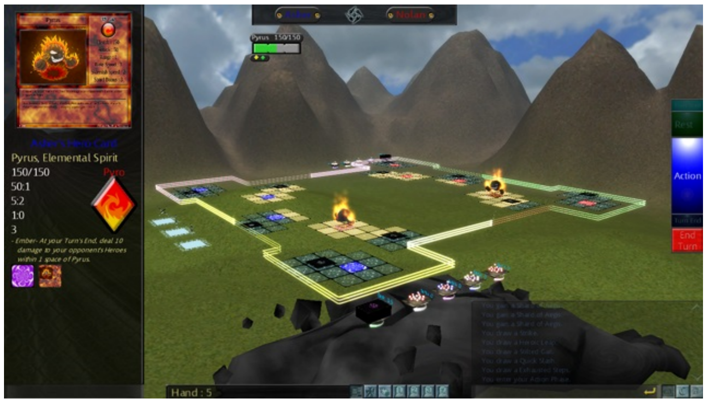


Figure 7: "Second Advent," an online collectible card game

By his senior year, having built a handful of other games, Eric wanted to build a larger project that he could refine and improve. He created the concept for the character and outlined the gameplay he’d like to implement. By the end of the class, he had produced a prototype for *Xeero*, with a few working levels.

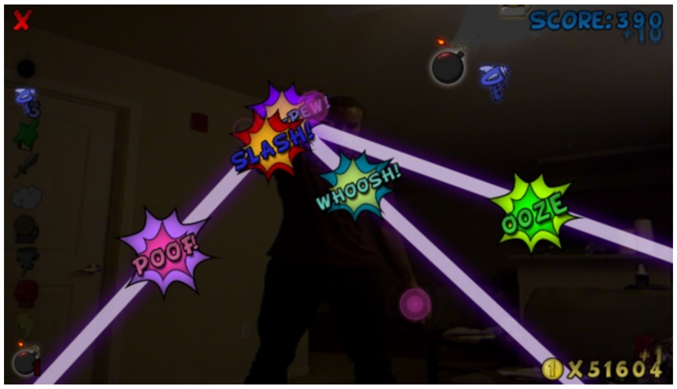


Figure 8: "Boom! Zap! Pow!" a game for the Microsoft Kinect

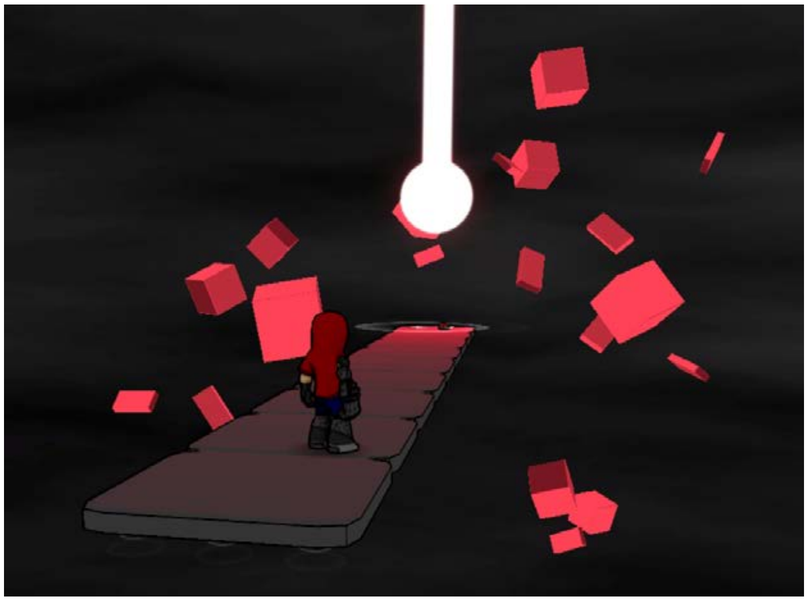


Figure 9: An early prototype of Xeero

After graduating, Eric continued to work on the project, refining what was already present and adding new features. The prototype created for UK covered the core platforming and digitizing mechanics, but Eric wanted to add combat, more digitizable objects, and more puzzles. He continued working on *Xeero* as a hobby as he started the graduate program Worcester Polytechnic Institute's Interactive Media and Game Development program.

Early in the program, he demoed the game to Professor Brian Moriarty, who encouraged Eric to pursue it as his Master's Project. He advised Eric to continue refining the game, to seek attention at conventions, and to pursue commercial distribution.

2.3.2. Forming the Team - Dan and Audio

When Dan and Eric met, they were both attending a training seminar for incoming teaching assistants for WPI. Both of them were new to the Worcester area, and didn't really know anyone. They quickly bonded as they were the only two students who would be working for the IMGD department.

A few months into our first semester, Eric pulled Dan aside and demoed *Xeero* to him. One thing that stood out in this current version of the game was an obvious lack of audio. There were no sound effects, no audio feedback, and no music. The game looked and played well, but needed a soundtrack. That was when Dan made a case for himself to join Eric's project.

Having grown up a trombone player, Dan always was interested in music. Throughout high school he had much experience in playing and performing music, but he had zero experience in writing it. During his undergraduate studies at RIT, he enrolled in a lot of music theory courses and took up creating digital music as a hobby. At the same time, he also took up an interest in game audio. He enrolled in classes that focused on recording sounds, and manipulating them using digital audio workstations (DAWs). Despite the experience of creating sound effects and music, there were few opportunities to actually use them in a game.

Xeero was the perfect opportunity to finally put the skills Dan had learned to the test. At the same time, it would be a chance to gain experience designing for one game specifically, instead of just generically. Eric agreed that he could use the assistance with the audio, so Dan became a member of the team.

2.3.3. Forming the Team - Anthony and Production

Anthony was always willing and ready to jump into a leadership role whenever possible. About a year into development, Eric and Dan were introduced to Anthony through Professor Moriarty after one of his spring classes. Anthony was returning to WPI for the IMGD graduate program working on the management track. Anthony, having been afforded the opportunity to take on leadership roles in multiple projects beforehand, had served as the producer on his MQP, *Demon Dissension*, with Nick Konstantino, Brian Seney, and Mike Metzler the previous year.

Professor Moriarty informed Anthony that Eric and Dan were in need of a producer for *Xeero* since the project was beginning to ramp up into a full-fledged project. At the time, Anthony was still looking for a project to work on for his graduate degree and *Xeero* presented the perfect opportunity to work on a project that had the potential to turn into something special.

Xeero not only provided another opportunity to gain experience as a producer, but the potential to add a high quality portfolio piece to Anthony's resume. Upon agreeing to join the team, Anthony was granted access to the repository and he took time over the summer to begin planning out the development path the project would take.

3. Design

Eric was responsible for the concept and gameplay of *Xeero*. The following section describes the concept, story, and mechanics for the game finalized by the writing of this document. The section will describe some of the "game objects" implemented for *Xeero* as well as the design decisions and motivations for specific objects.

3.1. Game Overview

Xeero is a 3D, third-person action-puzzle-platformer, in which the player controls an avatar, Xeero, to progress through levels. The player has a small set of actions the player can perform: running, jumping, attacking, dodging, and they must use all of these actions to solve puzzles, reach platforms, and defeat enemies.

The game is designed for Windows, with gamepad support. Through the underlying architecture of the game, it can be easily ported to the Xbox 360, and releases for Linux, Mac, PlayStation 4, and the Xbox One are possible with relatively few changes to the codebase.

3.2. Inspirations

Xeero draws from different genres (action, platformers, and puzzle) and draws specific inspirations from games within each genre. The game seeks to distill the essence from each genre and unite their mechanics in such a way that a single mechanic doesn't apply to just a single section of *Xeero*, but uses its properties to apply it to the entirety of the game's action-puzzle-platforming spectrum.

Figure 10: Combat from Darksiders (Yablonski) *and* Dust: An Elysian Tail (Chambers)

The game drew inspiration from action games, like Microsoft Studio's *Dust: An Elysian Tail*, Nordic Games' *Darksiders*, Nintendo's *Bayonetta*, and Sony's *God of War*. These 3D games offer fast, reflex-heavy combat encounters. They usually pit a single character, controlled by the player, against multiple enemies of varying types. Different enemies require different strategies; many encounters can require multiple attempts before the player is able to complete them, or new enemies may require the player to change their strategy.

The examples mentioned are exclusively in third-person perspective, lending to flashy, vicious combat animations, which can be satisfying to a player to achieve with a few button presses. These games focus heavily on "combos," where the player can chain different types of attacks together by pressing different buttons in different order or by holding certain buttons during the attack animations. Some games, like those mentioned above, also offer minimal platforming, usually giving the player a basic jumping or climbing controls, and allow the player to perform simple platforming tasks to break up successive combat encounters.

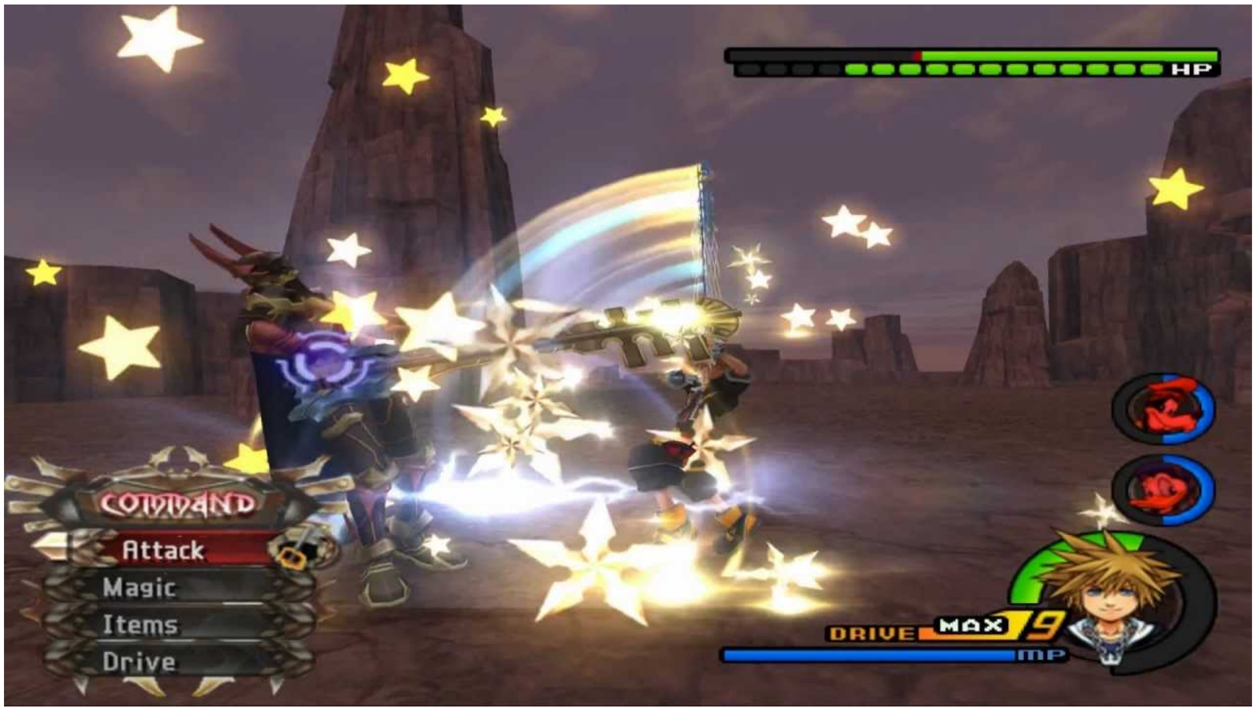


Figure 11: Combat from Kingdom Hearts II (venomblade891)

In a similar vein, *Kingdom Hearts* series, published by Square Enix, heavily influenced the design of combat. This game combines the combat of action games with the character progression of role-playing games. This game also demonstrates one of the "moments" of gameplay *Xeero* attempts to encapsulate: unlocking exciting new abilities. In games of this series, initially the player is presented with a character with a very minimal action set. As the player progresses in these games, they are given new abilities as their character "levels up." These abilities are designed to make combat experiences easier, either by giving the player more offensive options (through new kinds of physical attacks or special attacks/spells) or giving the player more options when on the defensive (through unlocking an ability that lets the player block, dodge, or escape enemy "combos"). Late in the game, the player is presented with the "flashy and vicious" combat animations that action games usually give players immediately; by having the player work to "earn" these abilities through leveling up and fighting several encounters, it is satisfying for the player when they receive a new, flashy ability that makes combat easier.

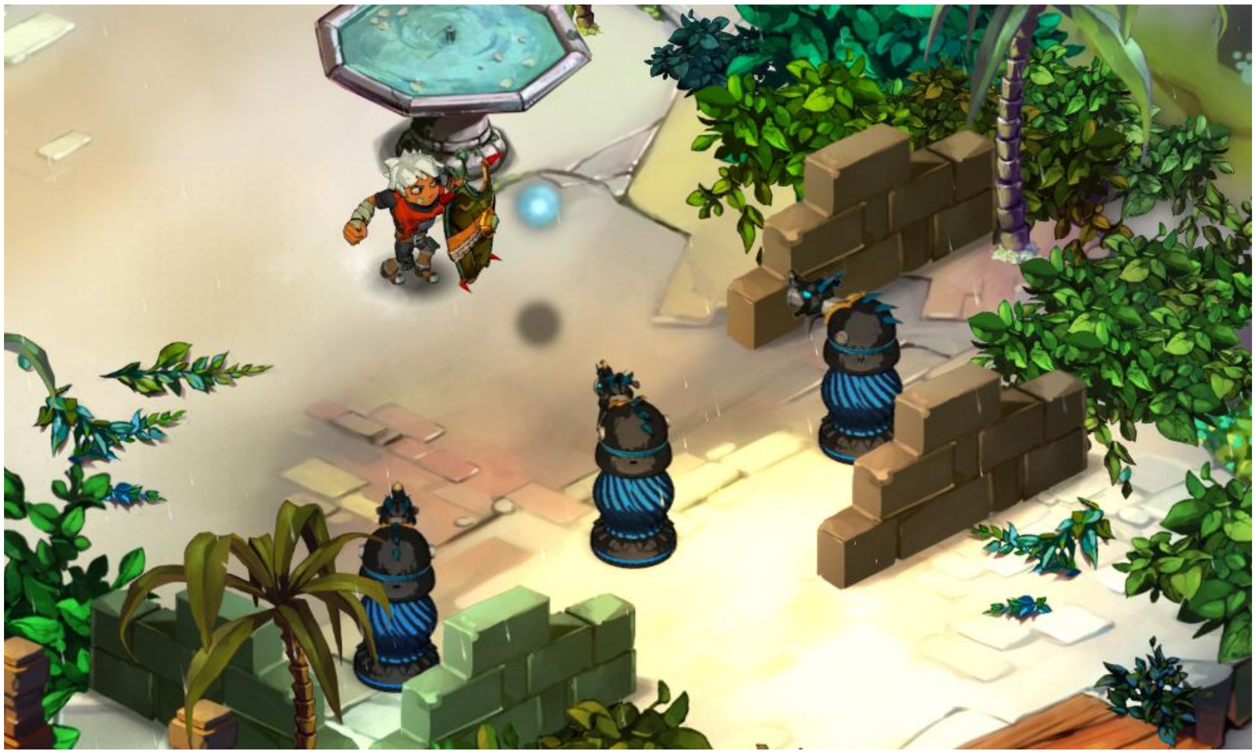


Figure 12: Blocking and countering in Bastion (C.)

Additionally, Warner Bros. Interactive Entertainment's *Bastion* provided inspiration for combat. In *Bastion*, the player has access to a variety of equipment, which perform different actions on button presses, and the player is allowed to swap them out. The game allows the player to play offensively *or* defensively, for example, the player can block attacks at the right time to counter the attack, damaging the attacker. Thus, in any combat encounter, the player has several options for how they choose to dispatch enemies, and the player is allowed to explore different play styles to find what is most satisfying for them.

The game also drew inspiration from puzzle games like Valve Corporation's *Portal* on how to construct interesting puzzles and how to teach players to play the game *while* the player is playing the game. The *Portal* series, a first-person puzzle solving game with *no* combat, has a very minimal set of actions the player can perform (the controls of the game essentially contain three buttons), but the game incrementally adds new objects in the world on which the player can use their actions, and each object adds a new level of complexity to the mechanics to which the player already has access. Each puzzle of the game is itself a tutorial for future puzzles in the game, and the game is steadily able to introduce higher complexity to the player.

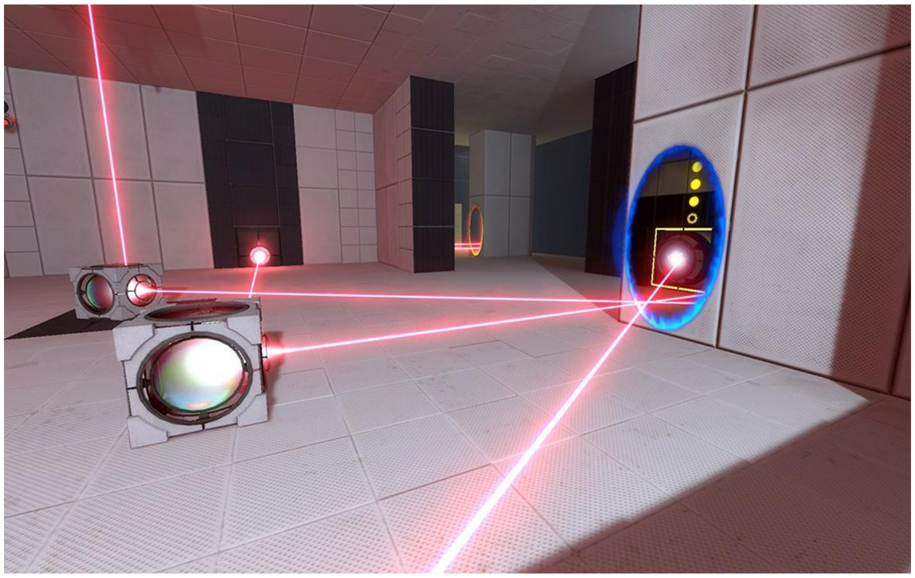


Figure 13: An example of the complexity of puzzles in Portal (Valve Corporation)

In these games simple objects have a variety of uses, and this game typifies the other "moment" of gameplay *Xeero* attempts to create: the interplay of game mechanics. The player can solve a complex puzzle consisting of several parts by using their limited actions and leveraging the behaviors of various objects in the world.

Puzzle games like *Portal* and action games like *Darksiders* have vastly different mechanics, gameplay, and target audience. So, the design of *Xeero* drew inspiration from games like Nobilis' *Trine* for combining different types of play experience. Games in the *Trine* series are action-puzzle-platformers, in which the player has control over three different characters, each character typifying one of those genres (i.e. there is a character designed for combat, one for solving puzzles, and one for platforming). The player is able to switch between the characters at will for whichever task is required of them. Then, short sections of each level are designed to either be devoted to one type of action or another or require a combination of characters.

Additionally, characters are not solely devoted to one style of play (e.g. the puzzle solving and platforming characters can also be used in combat, and the platforming and combat characters can also solve puzzles). In much the same way, *Xeero* seeks to unite different, and sometimes contradictory, aspects into a cohesive whole.

3.3. Story

The development of the story for *Xeero* occurred late in the project; the main character, world, and visual style had already been developed before the story was considered. Gameplay and visual style had been given precedence in development, and the purpose of the game was to deliver a gameplay-driven experience to the player. Despite this, story for this game was considered an essential component to providing a full experience for the player. Thus, a story was constructed to fit within the visual themes presented and to give context to the environment.

The following sections will describe the setting, characters, and plot for the game.

3.3.1. Setting

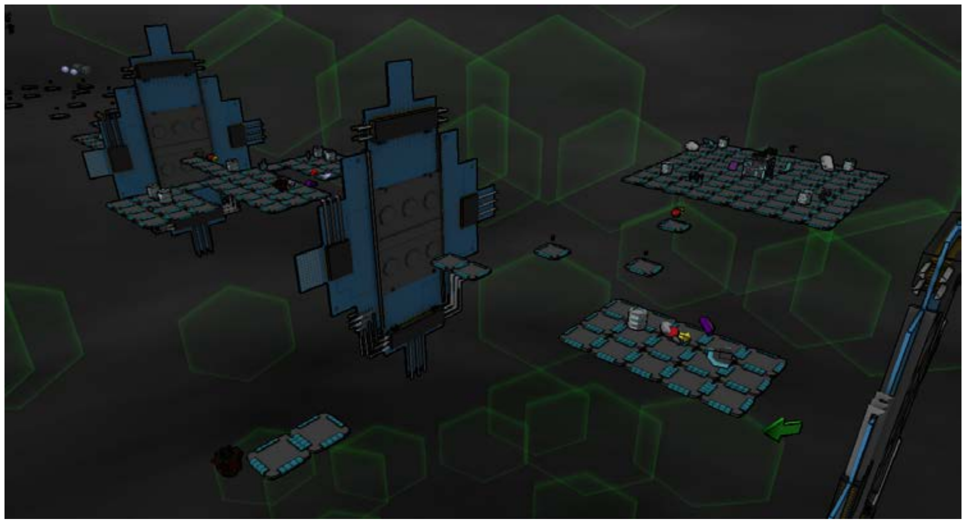


Figure 14: Overlook of a sample level in Xeero

Xeero takes in a single, infected computer, ostensibly the computer of the player playing the game. The world of *Xeero* is void-like, a mostly empty space composed of the remnants of a computer system devastated by the Alpha Virus. The virus worms its way through programs, leaving behind bugs that corrupt data in the system and malware that impede attempts to eliminate bugs.

The protagonist, Xeero, navigates through “hubs” in the game, subsections of the system off of which programs branch. By repairing the bugs in each program, Xeero can fix the corruption in hubs, and slowly work to repair the entire system.

3.3.2. Characters

Xeero

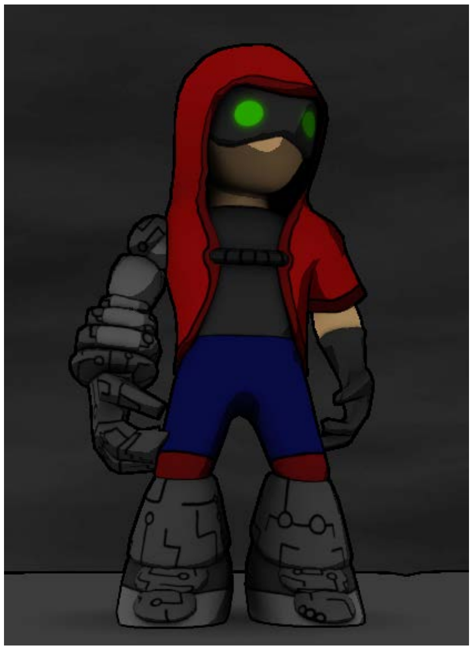


Figure 15: The titular character

Xeero, the protagonist of the game, is the character controlled by the player. Xeero is an anti-virus program, a program created specifically to seek out and destroy the Alpha Virus.

Xeero is determined and driven to find the virus (as dictated by his programming) and is silent (as he has no mouth). However, the program isn’t immune to distraction and is far from stoic.

He is outfitted in a red hood and a goggle-like terminal, through which he can emote with commonly used text emoticons. His right arm is a gauntlet, the tool that allows him to digitize aspects of the world around him, and his boots allow him the mobility to run and jump through cyberspace.

Xeero himself is an invader in the system he defends, injected externally by the player, and must defend himself against both malware and the system's security.

(Currently Unnamed Program)

This program is a companion program to Xeero, one also designed to hunt and destroy the Alpha Virus. Equipped with similar hood, boots, and gauntlet, and wielding the same digitizing ability as Xeero, she shares his drive for the mission for which they were designed.

Dead set on her mission, she is unafraid to make her own path, separate from Xeero, to hunt their prey. Xeero, however, prone to distraction and hopelessly smitten, makes finding *her* just as important to his mission as finding the Alpha Virus.

The Alpha Virus

This program is the driving force behind the events of the game. A virus of unknown origin, this creature infects systems, wreaking havoc and destroying data.

A hulking, massive creature, the virus emits corruption. The virus has the ability to corrupt anything it touches, leaving behind bugs in programs. It can replicate and mutate, creating hordes of different malware to thrash the system.

To the player, it is the reason their computer has been razed. To the anti-virus programs, it is their (mostly) singular reason for existing.

3.3.3. Plot

The plot of *Xeero* will be broken into segments, each segment will be delivered as part of the content distributed in one "episode" of the game. The story encompassed by the first segment, the segment this project is designed to create, introduces the player to the main actors in the story, and establishes the Alpha Virus as a threat.

At the beginning of the story, once the player starts the game, the Alpha Virus can be seen “downloading” itself into the player’s computer, corrupting and distorting the player’s display. The player must then run “Xeero.exe,” ostensibly an anti-virus program, to eliminate the Alpha Virus.

This leads the player into the world of *Xeero*. They follow the freshly-compiled protagonist as he drops into the infected system. With barely enough time to get his bearings, he is immediately confronted by the Alpha Virus. The massive creature crosses Xeero’s path, but the anti-virus program is so insignificant to the beast, Xeero doesn’t even catch its attention. Before long, the creature departs, off to spread corruption and mayhem, leaving Xeero alone to ponder his chances against the behemoth.

Not long after, Xeero witnesses his companion program chasing after the beast, oblivious to Xeero, intent on fulfilling her mission. Xeero is instantly smitten and resolves to find the Alpha Virus, if that means that he can find the girl. Here, the player starts the game, entering the tutorial, and joining the world of *Xeero*.

3.4. Level Structure

The following describes the different types of levels in *Xeero* and their intended purpose. This section will briefly describe some of the levels implemented for this master project.

3.4.1. Overview

The settings of *Xeero* are treated as an interconnected world for the player to navigate. The entire world of the game is a single computer system, composed of the infected programs the player can fix. The world is divided into “hubs” that connect the individual levels the player must navigate.

3.4.2. Programs

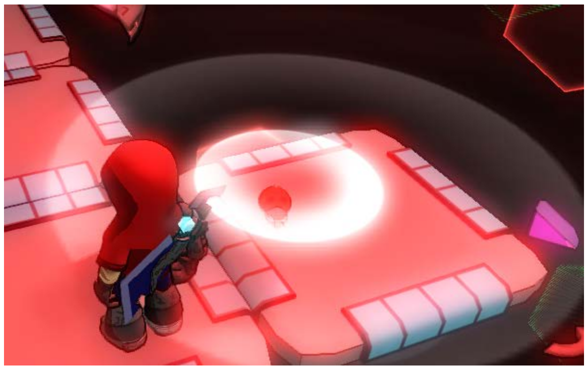


Figure 16: A "bug" in a program

"Programs," or levels, make up the bulk of playing time for the player. Within each program, there is a "bug" the player must reach. The bug is always located at the end of the level, and the player must sequentially move through all of the obstacles of the level to reach the bug.

Each level is designed to take 30-40 minutes to complete the first time the player attempts it, and is designed to be played all at once: while the player is allowed to "die" anywhere in the level to be returned to where they last failed, if the player completely exits the game, when they resume, they will need to start the level from the beginning again.

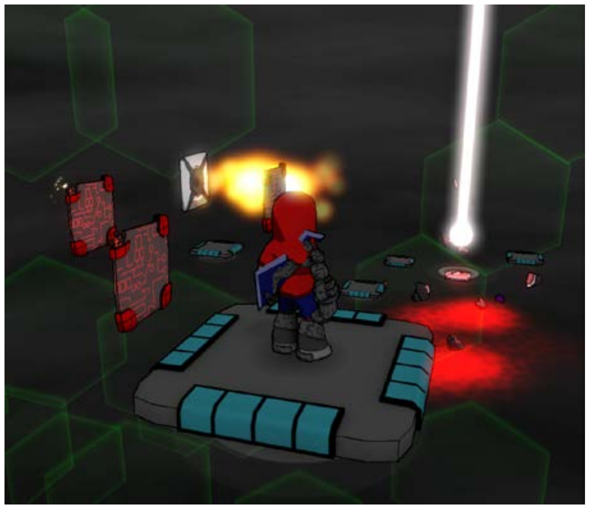


Figure 17: A pillar of light denotes the location of the bug

Each level in the game must be completed once to make larger progress in the game, but the player always has access to the programs and can play any level as many times as they want.

Each level contains currency (which regenerates each time the level starts) that the player can acquire to upgrade their character. Some levels contain special rewards for the player (like an upgrade or a container with a large sum of money) that the player can only collect once (i.e. after they have been collected for the first time, they stop appearing in the level). These special rewards may take more effort for the player to collect or may require an ability the player has yet to gain. For these, the player may want to revisit previously cleared programs to collect these rewards.

Once the player reaches the end of the level and stomps out the bug, they are transported back to the hub level to which the program is connected.

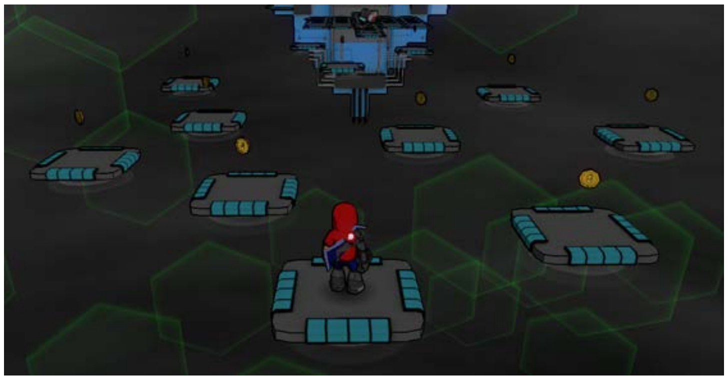


Figure 18: Money scattered around a level

3.4.3. Hubs

Hubs act as branching points for levels in *Xeero*. The entire experience of the game is broken into installments, and each installment has its own singular hub and levels connected to that hub. From any hub, the player can travel to programs in that hub or to adjacent hubs. Typically, the programs within each hub are blocked and can only be unblocked when the player destroys bugs in previous programs. Thus, each level of a hub unlocks a later level, leading finally to the last level of the hub, a boss fight.

Hubs, the levels within them, and the boss fight, are designed around a theme, allowing the visual layout of hubs to vary significantly from installment to installment. Each hub is also focused on introducing new digitizable objects and allowing the player to discover the various uses the object provides.

Hubs act as an area of both respite and challenge for the player. Hubs are free of combat encounters, and the player is able to travel to sections of the hub unimpeded by most hazards. Additionally, in every hub there is a station for the player to purchase upgrades and improve their character. However, platforming or puzzle-solving is required to reach new programs within the hub. Further, hubs usually contain multiple special items for the player to collect (like one-time containers of money) that require significant effort to reach.

3.4.4. Breakdown of Hub 1

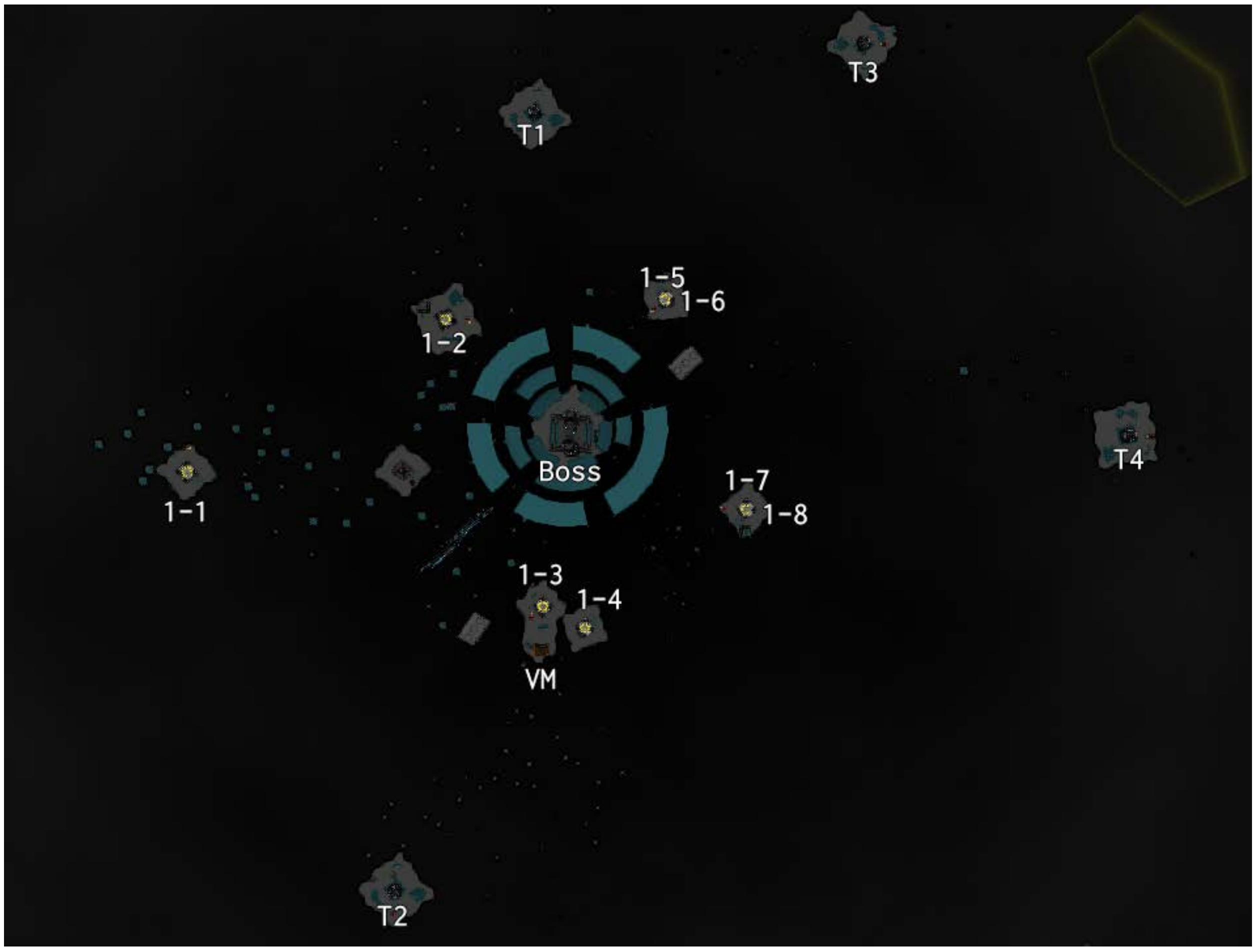


Figure 19: The first hub. "1-1," "1-2," etc. denote levels

Hub 1, the hub developed for this project, is based around the theme “debris.” The hub is composed of dozens of broken islands floating in space, with a swirling cloud of debris rushing past the player. The ultimate goal for the player is to reach the center of the hub, toward the giant portal leading to the boss fight.

There are eight levels in this hub (each 30-40 minutes long). Each level in the hub unlocks levels of subsequent numbering in the diagram (Program **1-1** unlocks **1-2**, and so on).

However, even after the levels have been unlocked, the player has to discover how to reach the portal to each level. Once all of the programs have been beaten, the large portal in the center, marked “Boss,” will active, allowing the player to fight the boss of the hub. At any time, the player can reach the station, marked **VM**, to purchase upgrades for their character (see section “**Objects - Vending Machines”**).

Additionally, there are special rewards, marked **T1** through **T4**, that the player can reach through challenging platforming (see section “**Objects - Big Archives**”). These give large amounts of money the player can use at the vending machine for upgrades.

3.4.5. Boss Fights

Boss fights act as the climax for each installment and are the final level the player plays in any hub. These are designed to allow the player to use the mechanics, abilities, and objects the player gained over the course of the hub in a combat situation. Each boss fight is related to the theme of the hub.

Each boss fight level also serves as a connection between hubs; after the player defeats the boss, they can travel through the level to reach the next or previous hub.

3.4.6. Boss Fight 1 - Beta Virus - Debris Construct

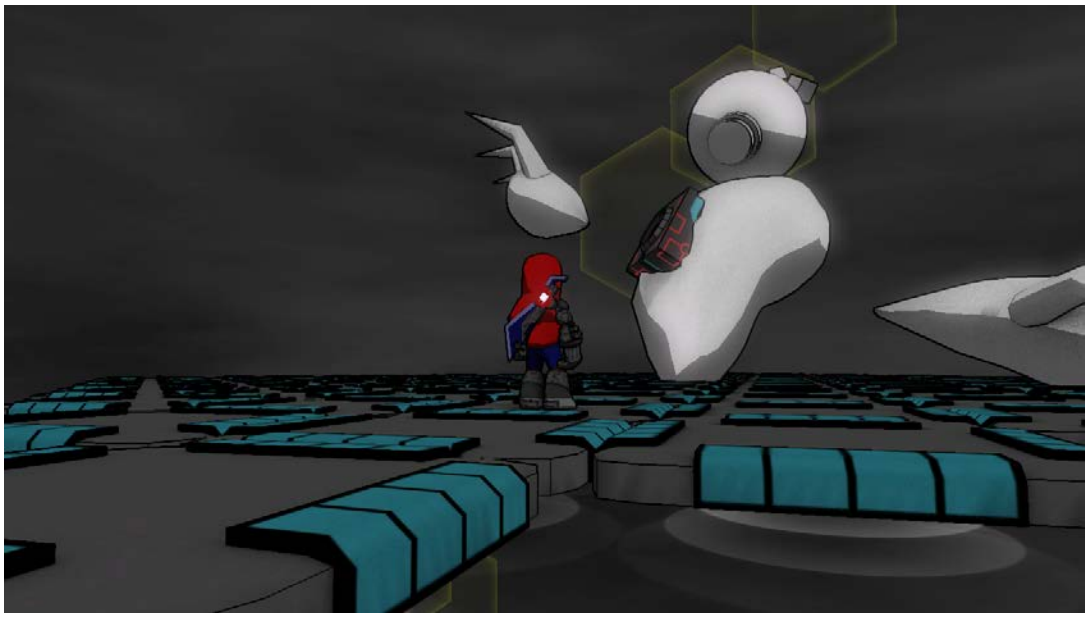


Figure 20: A mockup of the first boss in Xeero

The boss fight developed for this project, for the first installment of *Xeero*, was the construct boss, composed of scattered debris given life by the Alpha Virus. This boss is designed to test the basic combat maneuvers of the player, running, jumping, dodging, blocking, basic attacks, and leaping (see section "**Mechanics - Attacking and Defending"**). The boss also lets the player use their ability to digitize blocks and use properties of the object to defeat the boss (see section "**Digitizable Objects - Block**").

The player must use their dodging and jumping abilities to dodge the simple swipes of the boss's claws, and can use their ability to block attacks to deflect ranged attacks emitted by the boss. The boss incorporates a block object, lodged in its chest. The player can attack the block, dislodging it, allowing the player to digitize it. The player can then use the block to interrupt the boss's attacks (by forcing the boss to hit the block) to stun the boss enough to allow the player to damage it, eventually defeating the creature.

All levels in the game, whether they are programs, hubs, or boss fights, are designed to be modular. Very few large set pieces were developed for the game. Instead, the world geometry, puzzles and hazards are composed of smaller, modular pieces, each piece both altering the visual appearance and also the gameplay of the level. This allows for more rapid level mockups and development, and also allows more control and ease of use for players when the level editor is packaged and released with the game.

3.5. Mechanics

This section describes the how the game world is manipulated by the player, and describes the motivation for incorporating each mechanic.

3.5.1. Motivations

In many platforming games, and one of the games inspiring this project, *Portal*, the player is given a very small set of actions they can perform integral to type of gameplay (in some platformers, solely running and jumping, and in Portal, opening two types of portals and picking up objects). These games take their minimal set of actions and iterate, finding new, unique, and interesting ways the mechanic can be applied. In platformers, the player can have a variety of acrobatics based solely on jumping (wall jumping, climbing over ledges, stomping on enemies).

In *Xeero*, a game focused on blending different genres of games, takes a similar approach. It applies a single (or related pair) of action for each “slice” of gameplay (combat, platforming, puzzle solving), and attempts to find interesting ways each mechanic can be used and reused over the different play styles incorporated into the game.

In the game, for each “slice,” there is a pair of associated actions: running and jumping (platforming), digitizing and materializing (puzzle-solving), and attacking and defending (combat).

3.5.2. Running and Jumping

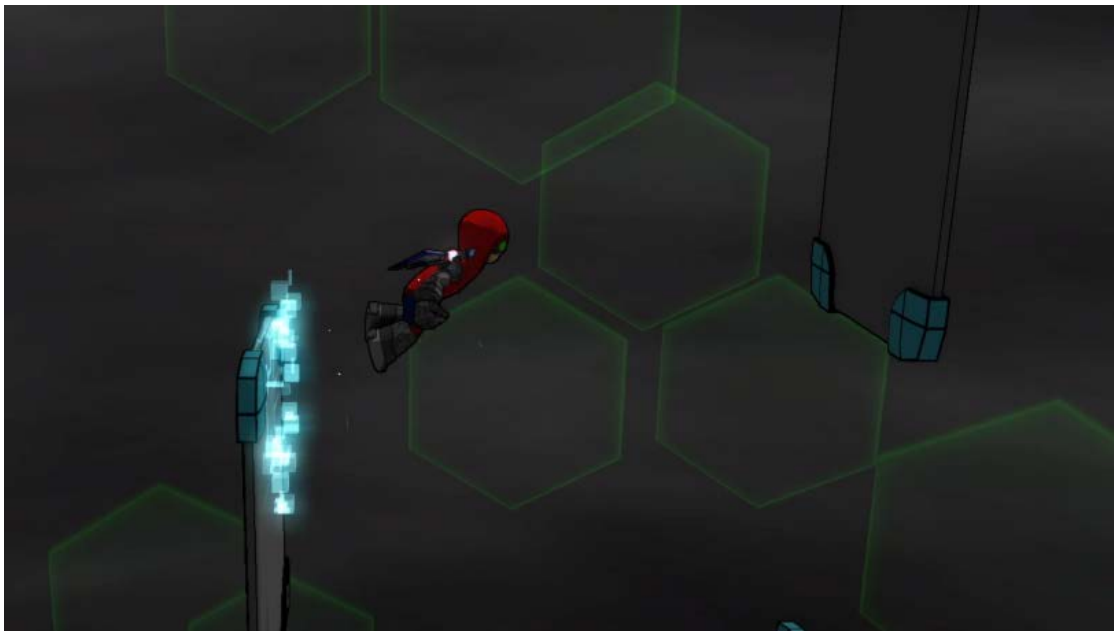


Figure 21: Xeero performing a wall jump

Running and jumping is a staple to many games. In platformer games, the focus of the game is executing properly-timed jumps or maneuvers requiring precise control of the character. In these types, jumping is reapplied in wall jumping or double jumping. These mechanics are essential in *Xeero*, with much of the challenge of navigating around the world consisting of properly timed or precisely executed jumps, double jumps, or wall jumps. Running/jumping can also be used to solve puzzles, or to give the player advantages in combat.

In the game, some switches act as pressure plates, and can only be activated by the player standing on them. In combat, enemies can be stunned if the player jumps on their heads, interrupting their actions and giving the player a moment to strike.

3.5.3. Digitizing and Materializing

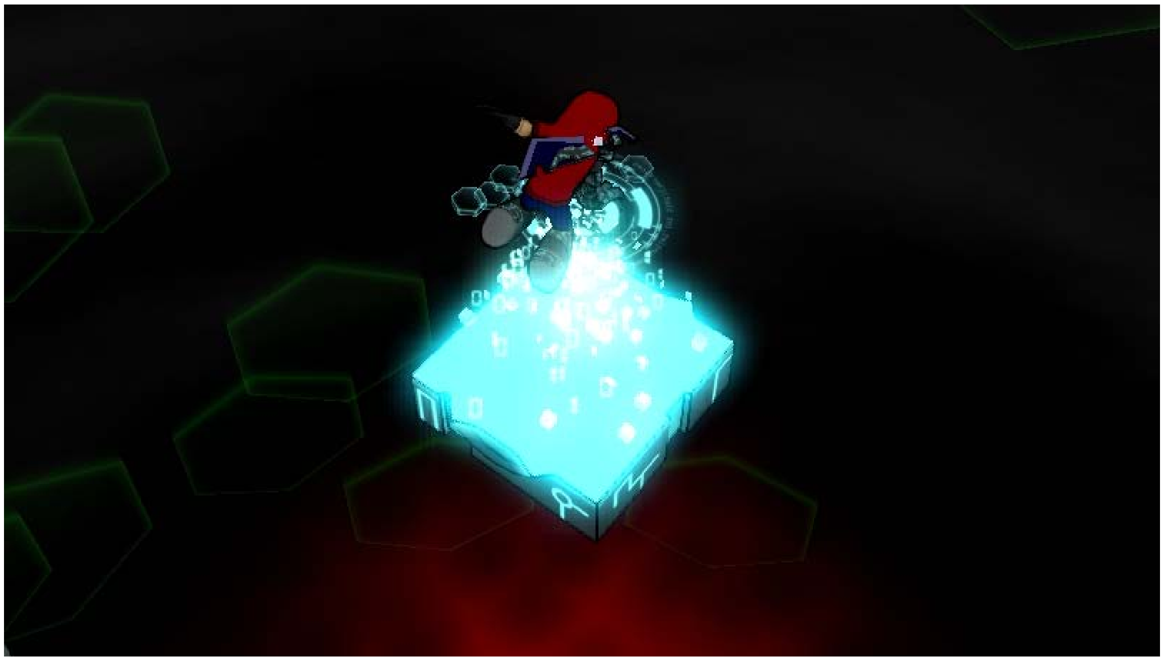


Figure 22: Xeero digitizing a block to create a platform for himself

Digitizing is the foundational affordance of Xeero, the ability through which nearly every other one of his abilities is acquired. He can digitize certain components of the world and store them in his gauntlet. Some objects he can reuse to help him navigate the world, while others give him new abilities.

The first type of object the player can digitize in the game are upgrade chips (see section: **Digitizable Objects - Upgrade Chips**). These are one-time objects; once they are digitized, they are permanently stored in Xeero's gauntlet. These give the player new abilities.

More commonly, the player can digitize various objects located around the world. These digitizable objects can be stored in the gauntlet and materialized back into the world when the player needs them. Each object has different effects, and the player can store up to two different objects in their gauntlet simultaneously. The player can digitize block objects to help the player reach cover long gaps or reach high platforms. They can also digitize bombs to destroy obstacles blocking the player's path.

Reusable objects like blocks and bombs can also be used in combat. Blocks, which help Xeero cross gaps, can also be used as blunt instruments. Xeero can materialize blocks into the world above enemies, and slam them down to quickly dispatch a large group of enemies. Bombs can be hurled into enemies, causing them to instantly detonate.

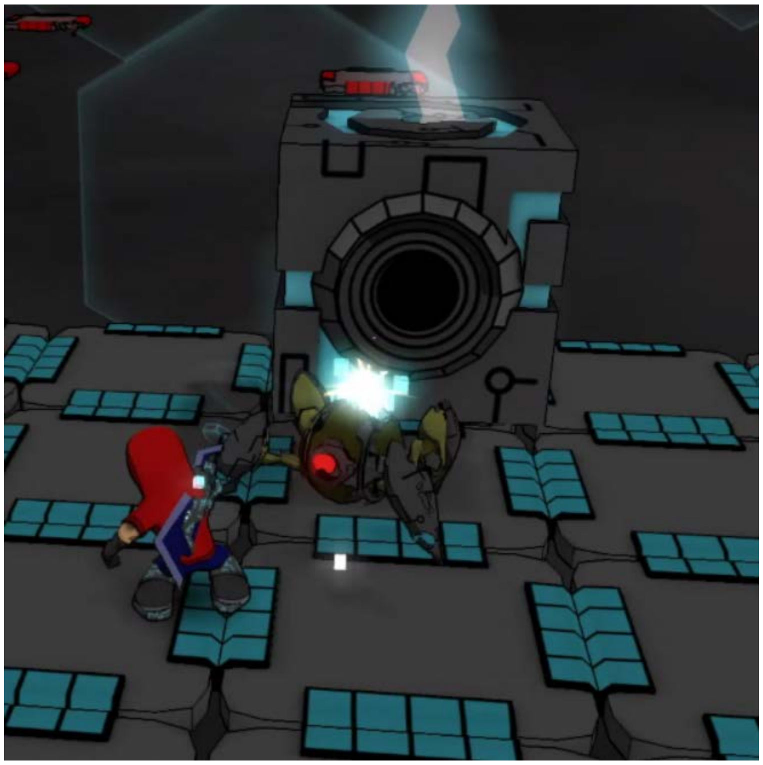


Figure 23: Using a block in combat to crush an enemy

3.5.4. Attacking and Defending

In the design of combat, focus was given to "depth and intention." Mike Birkhead, senior game designer for some of the *God of War* games, argues that successful action games focus on depth, rather than breadth. At any "state" for the player character, there are a relatively few number of moves they can perform, but each move leads to a state that allows another few moves to be executed, leading deeper and deeper until the character is able to perform a string of moves in succession. In action games, the focus is on the "combo," and allowing the player to string together long chains of moves in impressive fashion, evoking a sense of satisfaction in the player (see **Experience Goals and Audience**).

Almost in contradiction, Birkhead also makes the arguments for breadth of *intention* in action games. He argues that players have goals and intentions when playing action games; the goal is usually simple: “I want to defeat my opponent.” Intention is the question of *how*. Intention can be the strategy the player intends to employ: like using dodges and quick movements to flank enemies, committing to big, slow, devastating attacks to quickly clear out enemies, moving quickly and committing only to quick, light attacks. Birkhead argues that successful action games give the player a variety of viable intentions, allowing the player to play the game in the most satisfying way.

Birkhead also argues the value of constraints, or restricting the viable set of intentions to which the player has access based on the environment or enemies the player is facing. By introducing new environments or enemies that favor one type of intention over another, the designer can keep the player engaged by encouraging them to mix strategies, creating more interesting strings of attacks.

These philosophies were considered in the design of combat in *Xeero*. The game sought a relatively few number of commands dedicated to combat, but, combined with mechanics for platforming and puzzle-solving, it allowed for a breadth of intention from the player on how they chose to dispatch the malware before them. Additionally, at nearly any moment or “state” of the character, the player has access to most of their arsenal, allowing them to string together series of actions.

3.5.5. Reaper

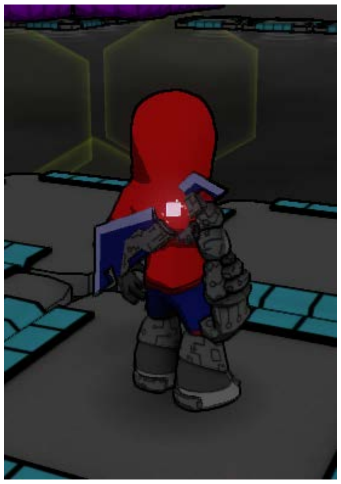


Figure 24: Reaper, attached to Xeero's back

Near the beginning of the game, the player gains Reaper, a telekinetic blade. This weapon follows the player for the remainder of the game, and grants Xeero many of his offensive and defensive abilities.

Offensive Abilities

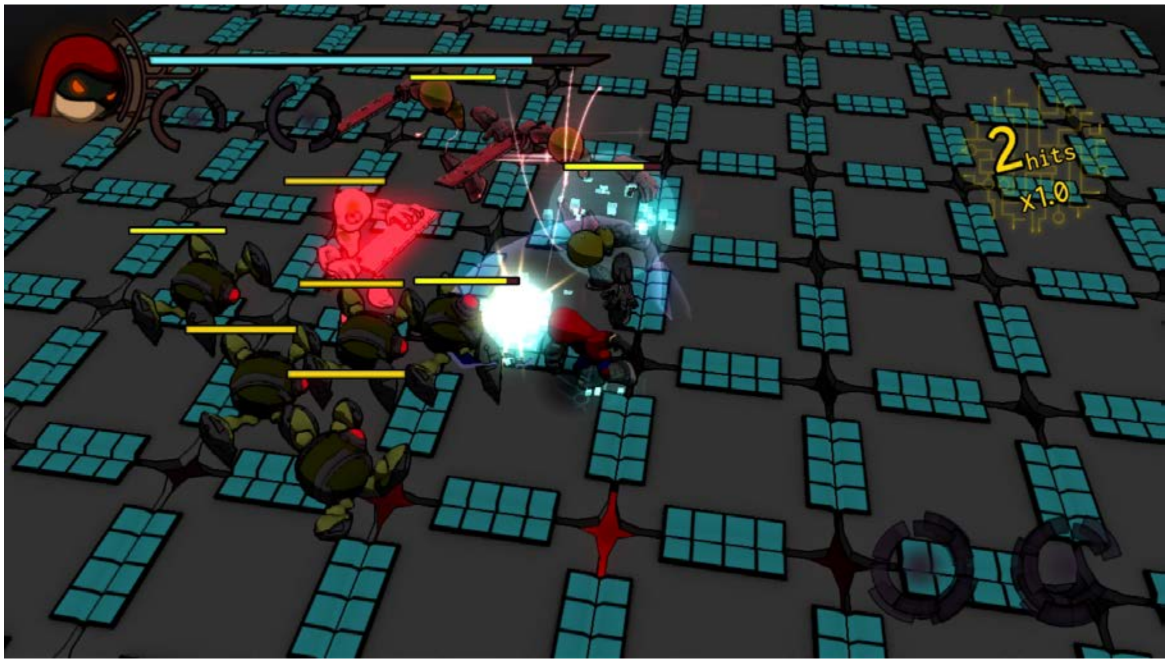


Figure 25: Xeero fighting a group of viruses

The player has access to two attacks, a primary and secondary, but each attack can be altered to create a larger range of options.

The primary attack is a basic attack, which is a combo of three swings the player can execute. This deals a moderate amount of damage, has a short range, and is the most common method of dispatching enemies.

The player also has access to a secondary attack, a “telekinetic leap,” through which the player can quickly leap to enemies, closing the gap between them and allowing the player to hit the enemies with their basic attacks.

The player can also use both the primary and secondary attack controls to apply different types of attacks. By holding the primary attack button, the player can use a “block-breaking” attack, a heavy attack that deals more damage and can stop enemies from blocking other attacks from the player. However, performing this attack ends their combo, restricting the number of actions the player can perform immediately after using the attack.

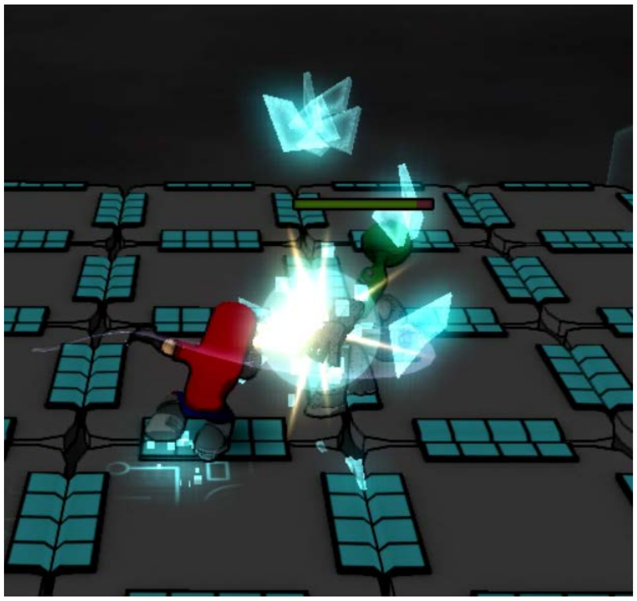


Figure 26: Breaking the block of an enemy

Similarly, by holding the secondary attack button, the player can perform a “knockback” attack, that deals little damage, but knocks enemies backward a far distance. This can be used to break up large clumped groups, give the player space to recover, or knock enemies off of ledges.

These attacks can be further altered through upgrades the player can acquire. Through upgrades, the player can acquire new attacks, but these attacks do not change the mechanism through which the player performs the attacks. Instead, the attacks are situational and are activated when appropriate. For example, the player can gain an attack that works as the final strike in their three-hit basic combo. This attack deals higher damage, but only to one enemy, while their default attack can hit multiple. If the player is faced with a single enemy, when the player reaches the last strike of the combo, the game will observe the number of enemies around and choose the most appropriate attack. Most of the attacks described (basic attacks, block breakers, knockbacks) can be altered in this way by acquiring upgrades.

Attacks were designed to be modified this way to keep the number of controls the player needed to learn low (allowing the game to reuse a mechanic in different ways) while providing a variety of flashy, satisfying moves.

Offensive abilities, like others in *Xeero,* can also be used outside of combat. The basic attack can be used to trigger switches necessary to solve puzzles. The telekinetic leap can be used to quickly close the gap between enemies *or* objects, so the player can use them to quickly hit timed switches. The player can also use the leap to cover large gaps the player couldn’t otherwise cross.

Defensive Abilities

Reaper also allows the player to block attacks. While blocking, all damage the player takes is dramatically reduced. However, the player is only able to block attacks from the front. The player can use the block to mitigate a barrage of damage coming at them, or if they are willing to take a risk, they can use block to deal damage to enemies. By initiating a block at the right moment before an attack hits, the player is able to counter the attack, reflecting projectiles and stunning attackers while dealing damage.

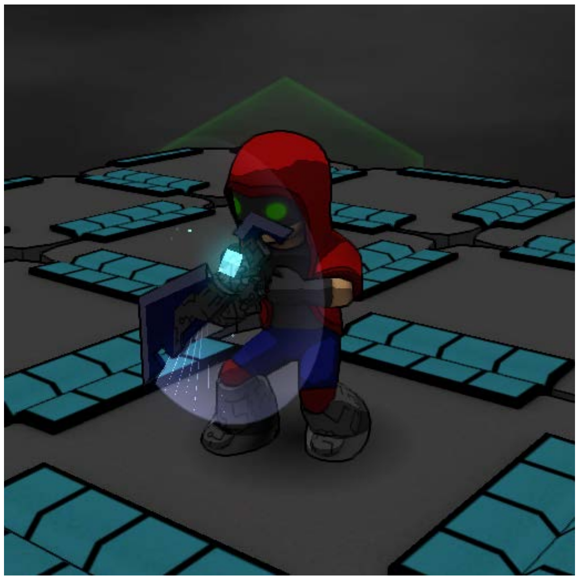


Figure 27: Blocking with Reaper

Additionally, the player can dodge. Dodging quickly moves the player in one direction and makes them temporarily immune from taking damage. The player can dodge incoming strikes, quickly engage or disengage from a group, or quickly move to flank an enemy.

Combat and Intentions

Through the dedicated attack actions, platforming mechanics, and puzzle-solving tools, the player has access a breadth of "intentions" they can have in combat. In combat encounters, the player can rely solely on the basic attack to aggressively clear out groups of enemies, use digitizable block objects to smash groups of enemies to quickly eliminate them, or play defensively, relying on the mobility afforded by dodging and the countering afforded by blocking to let enemies effectively eliminate themselves.

3.6. Controls

Breadth of intention was also considered when developing the controls for *Xeero*. Emphasis was given on allowing the player to use whatever control scheme they found comfortable. This manifested itself early in development with bind-able key-mappings for all controls, controller support for the PC version of the game, and both a left-handed and right-handed control schemes for the mouse and keyboard.

Many PC games are exclusively designed for right-handed players. The decision to by default assign left-handed controls (*in addition to* right-handed controls) was mostly a personal one; the designer of the game, Eric, is left-handed.

Many of the games inspiring *Xeero* were built for consoles, and the game's controls are most similar to third-person action games, so controller support was added for players who are accustomed to (or prefer) the controls of console action games.

Additionally, player feedback was vital in crafting the controls used by the game. Often usability testing revealed conflicts with the designer's intention for controls and player's expectations. New user-settable control options and control redesign emerged from player feedback.

3.7. Objects

The following section details how the design philosophies described in the **Mechanics** section were applied in the design of level elements in the game.

3.7.1. Platforms

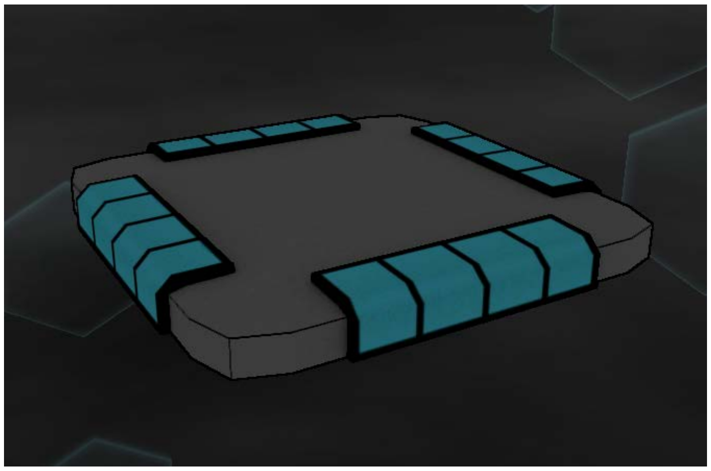


Figure 28: A static platform, a basic building block of levels

These are the building blocks of levels, and the primary means for the player to get from place to place. Xeero can stand, run, and jump from these, and can grab onto their ledges to pull himself up.

By themselves, they offer little challenge, but when they are spaced properly, or variants are used, they can introduce challenge for the player.

3.7.2. Small Platforms

These are identical to default platforms but smaller in size. The player has to more precisely aim their jumps to reach these platforms.

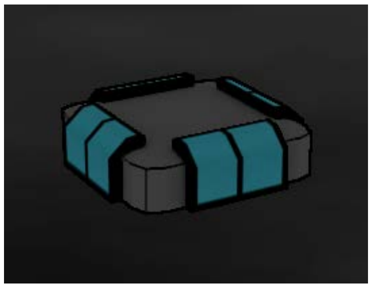


Figure 29: Small platform

3.7.3. Falling Platforms

These broken, corrupted platforms can't support weight for very long. Whenever the player (or any object) touches the platform, it shakes and falls away. These can be used in platforming to force the player to keep moving, or introduced in combat to add "hotspots" the player and enemies should avoid.

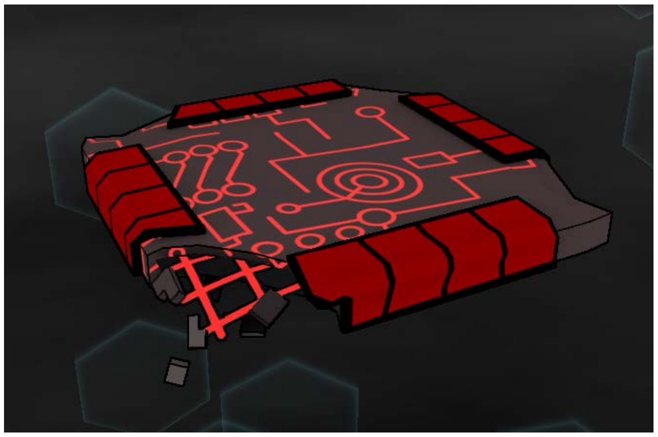


Figure 30: A broken, falling platform

3.7.4. Fading Platforms

These broken platforms only function some of the time. They oscillate in a cycle, from translucent to opaque. When translucent, objects fall or "phase" right through them. When platforming, the player must time their jumps correctly to avoid phasing through these.

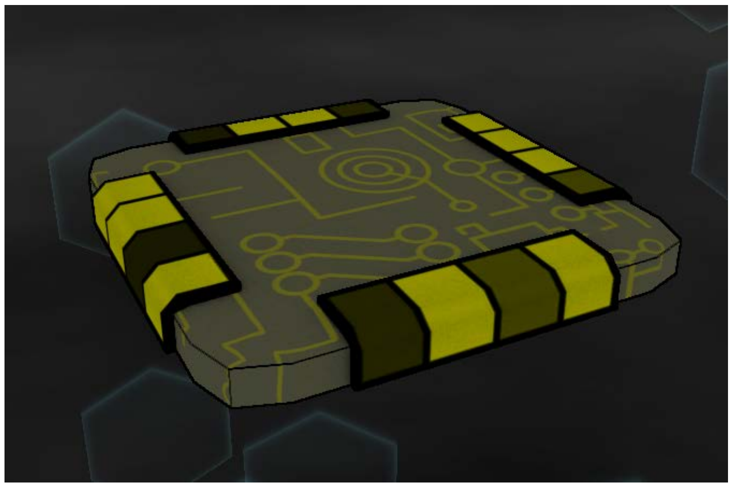


Figure 31: A broken, semi-tangible platform

3.7.5. Walls

These are used with one of the variants of the jump mechanic, the wall jump. The player can leap off of these walls to reach far away or higher platforms. If the player idles, they will begin to quickly slide off of the wall and eventually fall off. Like falling platforms, the player has limited time to correctly execute a succession of wall jumps. These walls provide variants like platforms: small, falling, and fading.

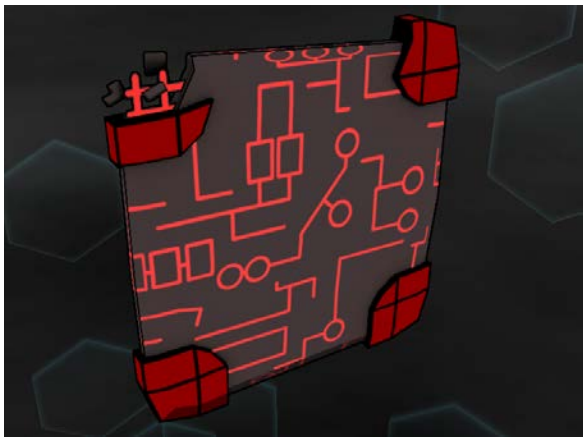


Figure 32: A broken wall

3.7.6. Breakpoints

These act as checkpoints for the player. As long as the player touches the breakpoint, the state of the level is automatically saved: the position of every object, the state of every switch. When the player "dies," the level resets, depositing the player back at the breakpoint, restoring the level to the same state as when the player touched it.

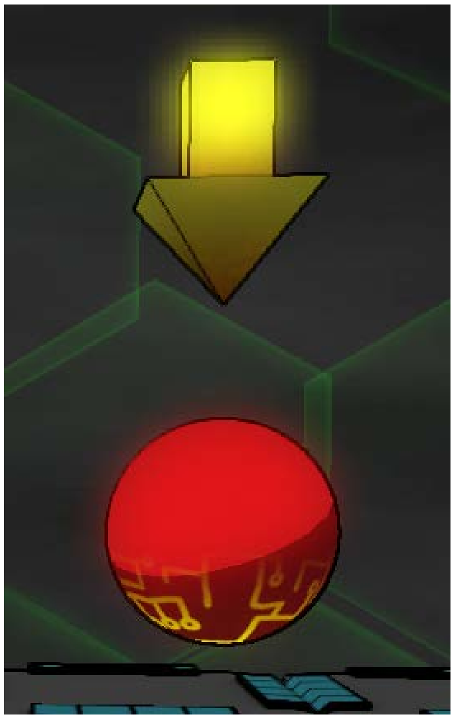


Figure 33: An activated breakpoint

These are essential to the "tough but forgiving" gameplay the design of the game attempts to create (see section **Experience Goals and Audience**). These are spread liberally throughout the levels to allow the player to frequently save their state.

3.7.7. Leap Targets

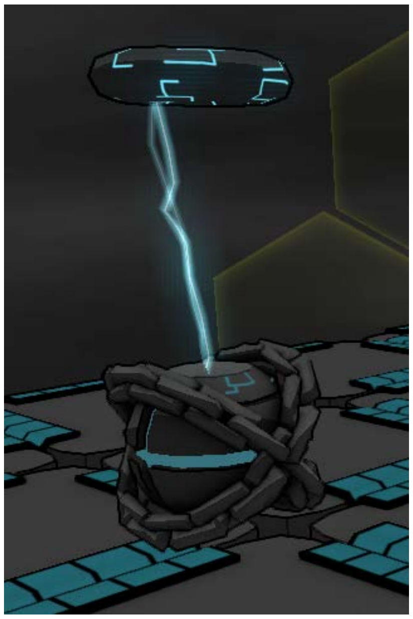


Figure 34: A tethered leap target

This object is designed to give platforming functionality to a combat ability (as discussed in the “Motivation” section of **Mechanics**). The player can use their telekinetic leap ability to reach this object from far away. This is useful for climbing large gaps the player otherwise couldn’t cross. It, like walls, requires quick reaction and more precise control from the player to execute successive leaps at these targets.

3.7.8. Digitizable Objects

These are objects that allow the player to frequently use Xeero’s ability to digitize. Different objects have different effects and purposes, and it is impressed upon the player to determine how to use the objects with which they are presented to solve puzzles or pass obstacles.

Blocks

Blocks are solid cubes, tough and sturdy, and not easily moved. The player can pick up and reuse blocks, which take up a slot in their gauntlet's inventory. Despite their simple nature, blocks have a variety of uses.

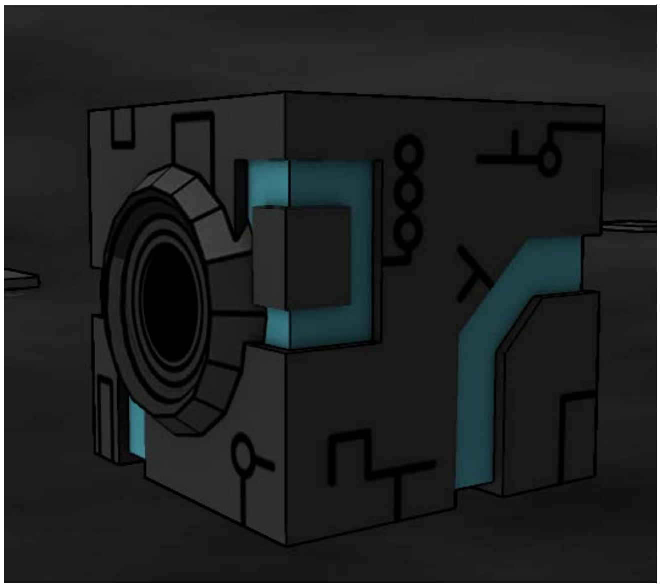


Figure 35: A digitizable block

Blocks can be used like platforms. When in the world, the player can stand on them and grab onto their edges. The player can materialize a block in front of Xeero to climb up to reach high platforms. If the player is in mid-air, the player can materialize a block as an improvised platform, allowing them to cross large gaps.

The player can also use a block's sturdy nature to create obstructions. Some objects prevent the player from progressing: **Gates** block the player's path, **Slicers** can move too quickly to be dodged, and **Compressors** crush any object that pass under it, but

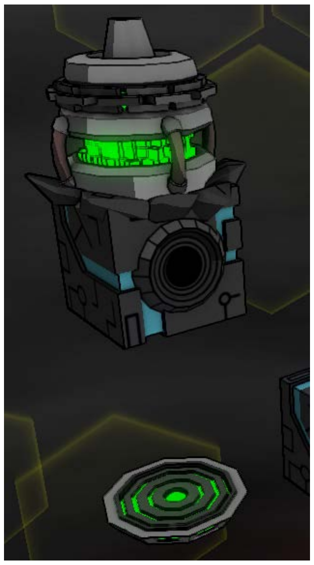


Figure 36: A compressor ledged in a block

the player can wedge a block under them, jamming them, and allowing the player to pass through unharmed.

The weight and shape of the block can also help the player solve puzzles. **Buttons** require the player or another heavy weight to keep them active, and **Block Switches** require a block to activate. The player can slam a block onto those switches, activating them to solve puzzles.

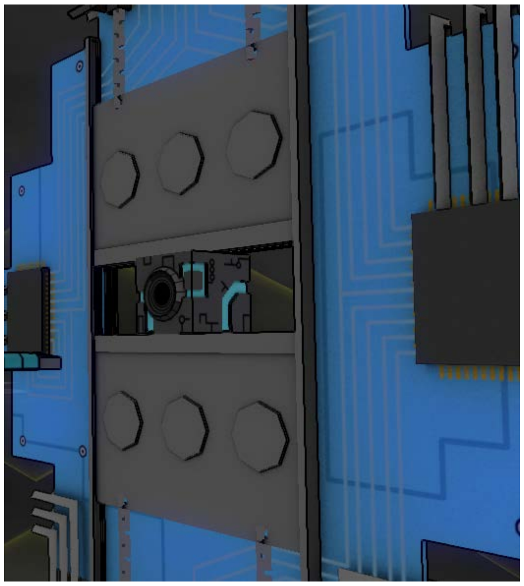


Figure 37: Using a block to wedge a gate open

Blocks also have uses in combat. As with buttons and block switches, the player can slam blocks onto target malware, crushing them and eliminating large clusters of viruses.

Blocks are the focus of the first hub of *Xeero*, and are the only digitizable tool of its type implemented for this project.

Upgrade Chips

These are the primary means for the player to acquire new abilities. The player can digitize these into Xeero’s gauntlet, permanently imbuing him with a new ability.

The abilities granted by chips can vary. They can allow the player to digitize a new type of object (like **Blocks**) or give the player a new platforming ability (like wall jumping - see **Mechanics - Platforming**) or a new combat ability (like the telekinetic leap - see **Mechanics - Offensive Abilities**).

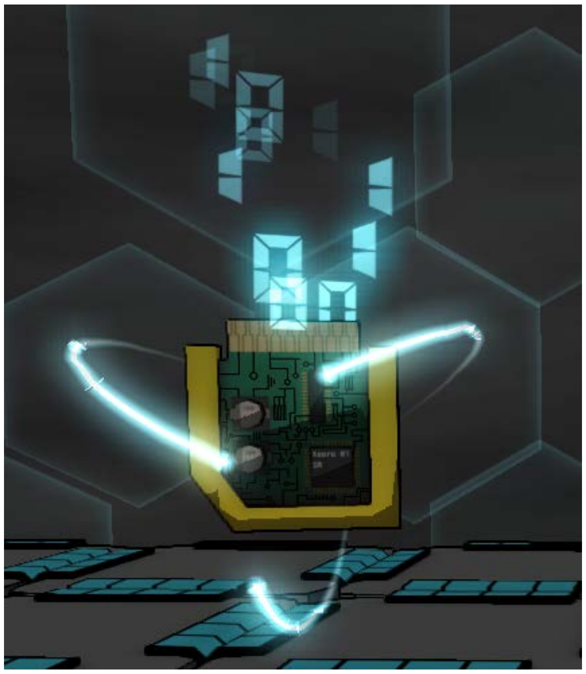


Figure 38: An upgrade chip

These can be scattered around levels or purchased from **Vending Machines**. When in a level, it is treated as a special one-time object; once it has been picked up and integrated, it will no longer appear in the level.

3.7.9. Hazards and Obstacles

These objects, usually components of system security, serve to impede Xeero’s progress. These offer puzzle-solving and platforming challenges for the player.

Gates

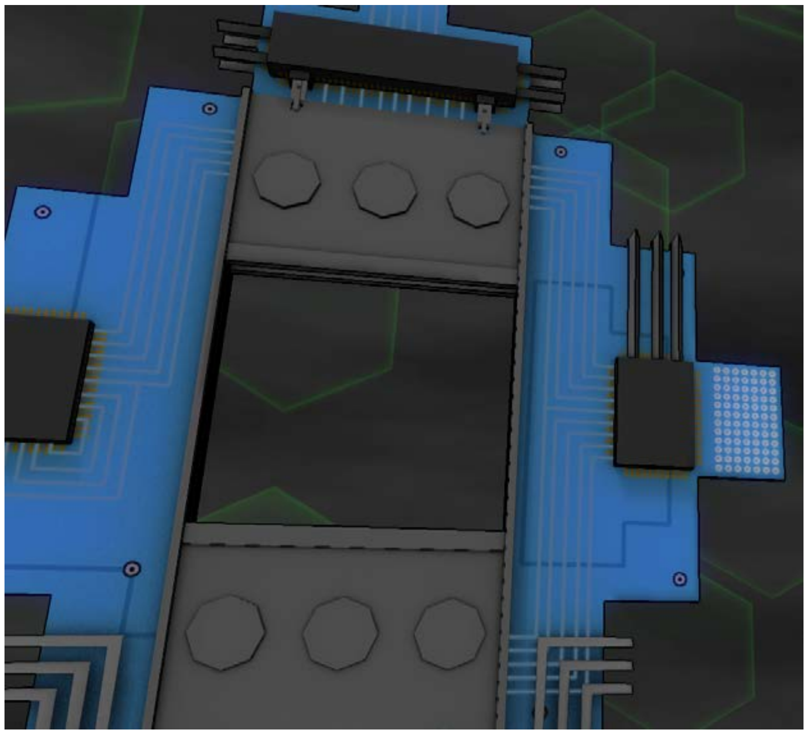


Figure 39: A gate, which can block Xeero's progress

These large gates block Xeero from proceeding until he opens them. Gates can only be opened by **Switches,** which may require the completion of a puzzle or platforming segment for access.

Bit Slicers

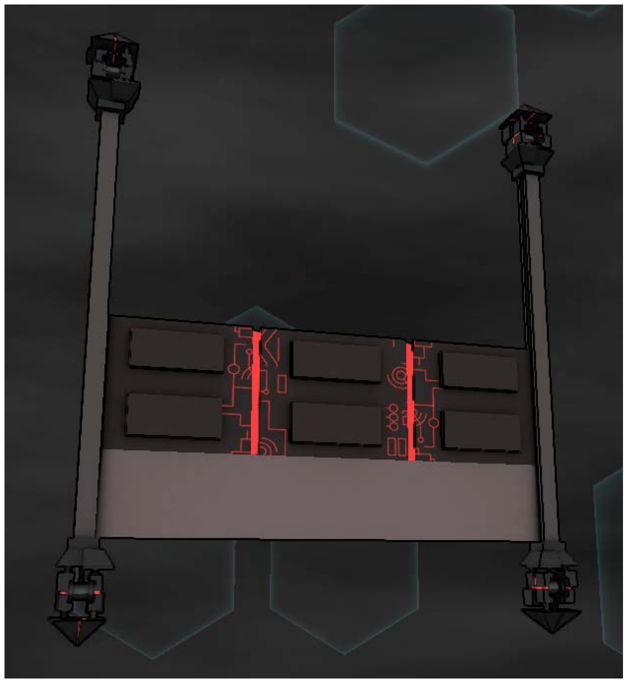


Figure 40: A bit slicer

These guillotine-like slicers rapidly slice up and down. They serve as a platforming challenge; the player must time jumps correctly to avoid getting sliced.

Slicers can also serve as a puzzle element. They can slice so quickly, it is *impossible* for the player to jump through in time. In such cases, the player must either obstruct the slicer with a **Block** or deactivate it with **Switches**.

Additionally, if there are **Enemies** nearby, the player can lead them through a slicer to quickly dispatch them.

Motion-Sensing Compressors

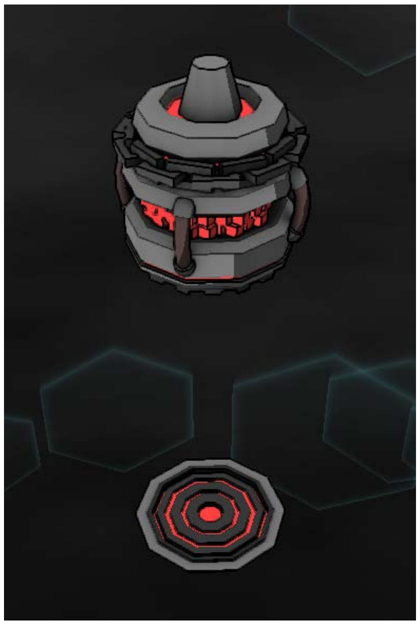


Figure 41: A motion-sensing compressor

These deadly obstacles immediately crush any object that passes beneath. The player can't move under it fast enough to avoid getting crushed, but must find a **Block** to obstruct or a **Switch** to deactivate it.

Corruption Walls

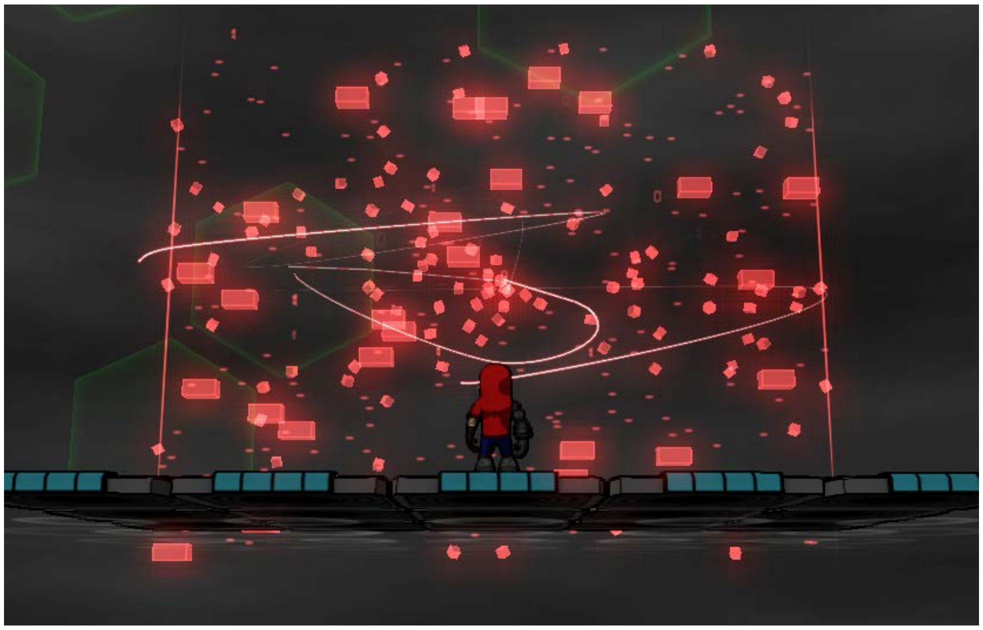


Figure 42: A corruption wall blocking Xeero's progress

These impenetrable obstacles block Xeero's progress. They knock back any object that touches it, and reflect all attacks. They usually appear near malware, and the corrupted data must be freed from malware before these can be disabled.

In combat, they also reflect the attacks of enemies, allowing the player to use the enemy's attacks against them, giving the player another "intention" in combat (see **Mechanics - Attacking and Defending**).

Turrets

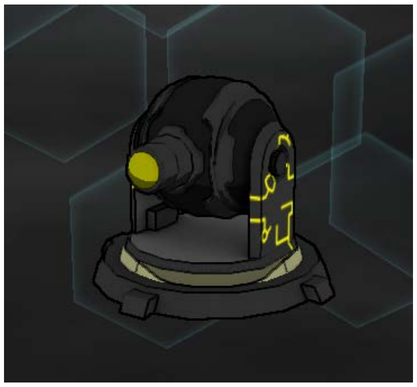


Figure 43: A laser turret

Turrets automatically target and fire lasers at Xeero. The player can dodge to avoid their assault. The player can also attack the turrets to destroy them, and, if they are far enough away, the player can block and counter their attacks to destroy them (see **Mechanics - Defensive Abilities**).

Purge Gates

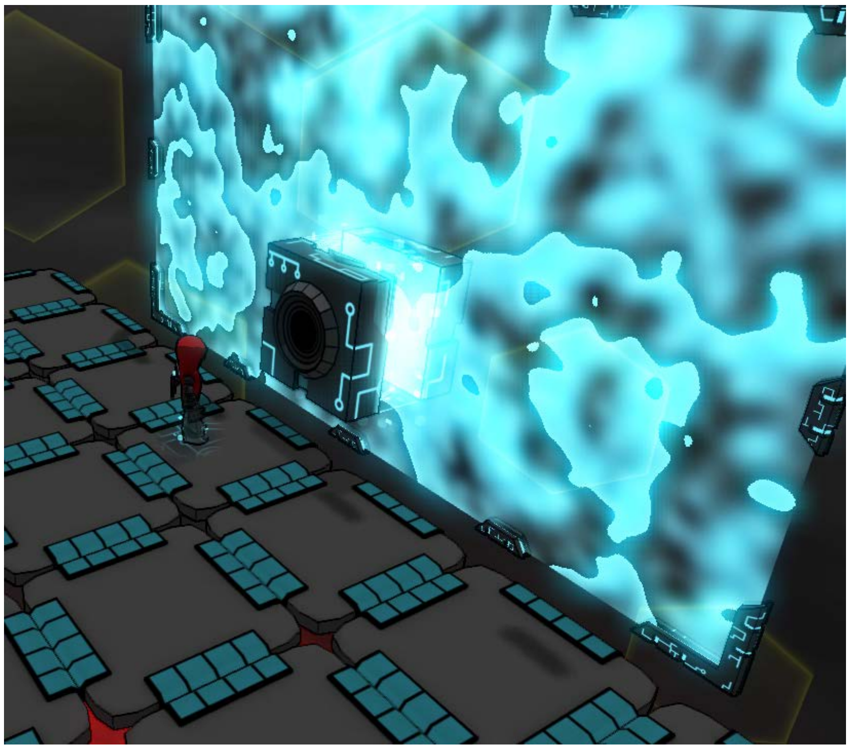


Figure 44: A purge gate purging a block

These gates don't physically impede Xeero's progress, but instead "purge" any digitizable object that touches the surface of the gate, whether they are materialized in the world or carried by Xeero in his gauntlet. If the player needs to transport a digitizable object from one area to another, they must either find a new way to reach the area or disable the gate with **Switches**.

3.7.10.Switches

Switches act as puzzle elements in the game. These are the "locks" for which the player must use a "key," or a specific type of action, to unlock. These can be linked to any receiver in the game (a receiver being an object like **Gates, Slicers, Compressors,** etc.) and can be arranged to challenge the player in many different ways.

Hit Switch

These are simple switches, triggered by a swing of the player's weapon. A hit switch can be activated as long as the player can get close to it. They can be placed behind large gaps, requiring the player to figure out how to cross the gap in order to reach the switch.

Hit switches can be linked with each other, requiring the player to touch every switch in the "circuit" to activate it. The component switches of a circuit can be scattered in various locations to increase the challenge.

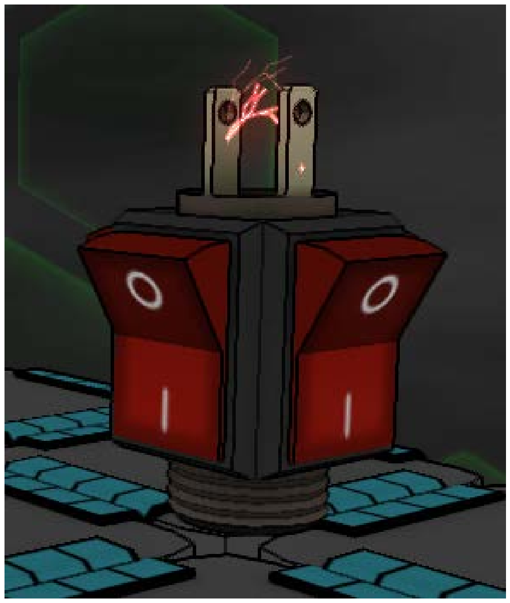


Figure 45: An active hit switch

Timed Hit Switches

These function similarly to **Hit Switches,** but when linked in a circuit, they can be set to a timer. Then, once one switch has been hit, the player has a limited amount of time to activate the remaining switches in the circuit. Hitting one switch in the circuit resets the timer on all of the timed hit switches. Circuits can be arranged to require the player quickly execute a set of actions or to use their telekinetic leap (see **Mechanics - Offensive Abilities**) to reach all of the switches in time.

Buttons

Buttons are triggered when Xeero stands on them. They can be set to either immediately deactivate when Xeero steps off, or to wait until a delay timer runs down, giving the player a limited period to perform a task before the switch fails.

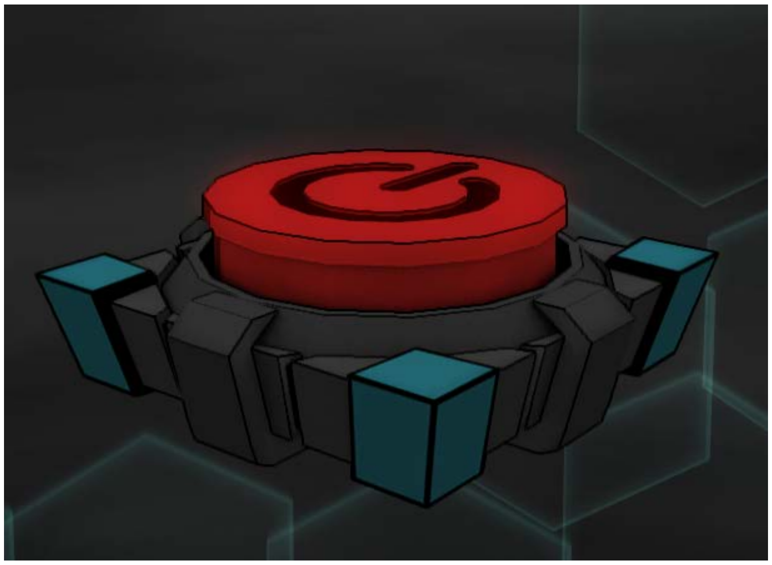


Figure 46: A button

Buttons can also be activated by placing a **Block** on them. Doing so eliminates any timed requirements, assuming the player can find a block to use.

Block Switches

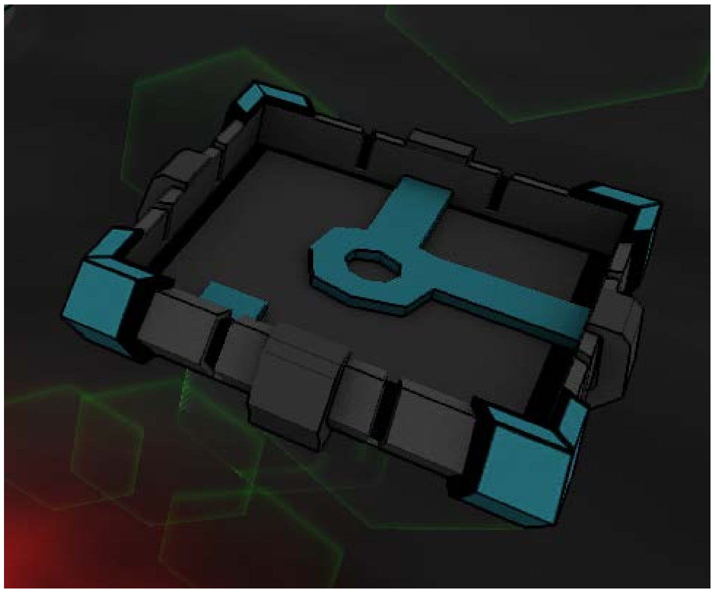


Figure 47: A block switch, waiting for a block

These function similarly to buttons, but can only be activated by placing a **Block** on it. They can be deactivated immediately or on a timer.

3.7.11.Corrupted Objects

These special hazards are corrupted versions of other objects the player can use (e.g. switches or portals). They act like **Corruption Walls**, knocking back any object that touches them, and reflecting all attacks. The player can’t interact with these objects until their corruption has been cleared. To do this, the player must enter the level of the bug that is corrupting the object and destroy it, freeing the object to be used.

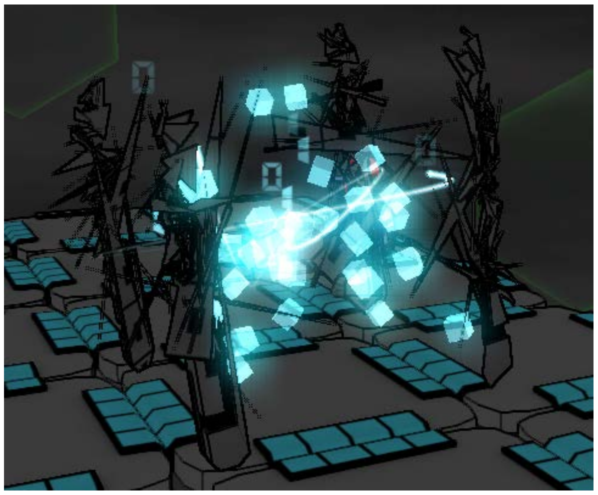


Figure 48: A corrupted object spawner

Corrupted objects are used primarily in hub levels, helping to tie individual levels into the larger puzzle of the hub. Often, to reach the final level (the boss fight) of the hub, the player must use the objects scattered around the level. These objects may be corrupted, requiring the player to clear levels to gain their uses. Additionally, other rewards may be scattered around the level requiring the use of the corrupted object to obtain them.

3.7.12.Enemies

These creatures are malware introduced to the system by the Alpha Virus. Early in the game, the player is given a weapon they can use to fight the creatures and free their corrupted data.

Combat encounters occur within levels, usually in large spaces. When the player approaches certain areas of the level, these enemies materialize and combat begins. In some cases, the player must defeat all enemies to progress; in others, combat is optional. Combat encounters, platforming, and puzzle-solving are interspersed throughout each level to give the player an even mix of each.

Malware in the game can take many forms, but the majority of enemies in the installment developed for this project are viruses. Viruses are not too difficult to defeat one-on-one, but they have the ability to replicate themselves, quickly overwhelming the player. Each virus is designed to encourage the player to discover and exercise different strategies and moves (see **Mechanics - Attacking and Defending**).

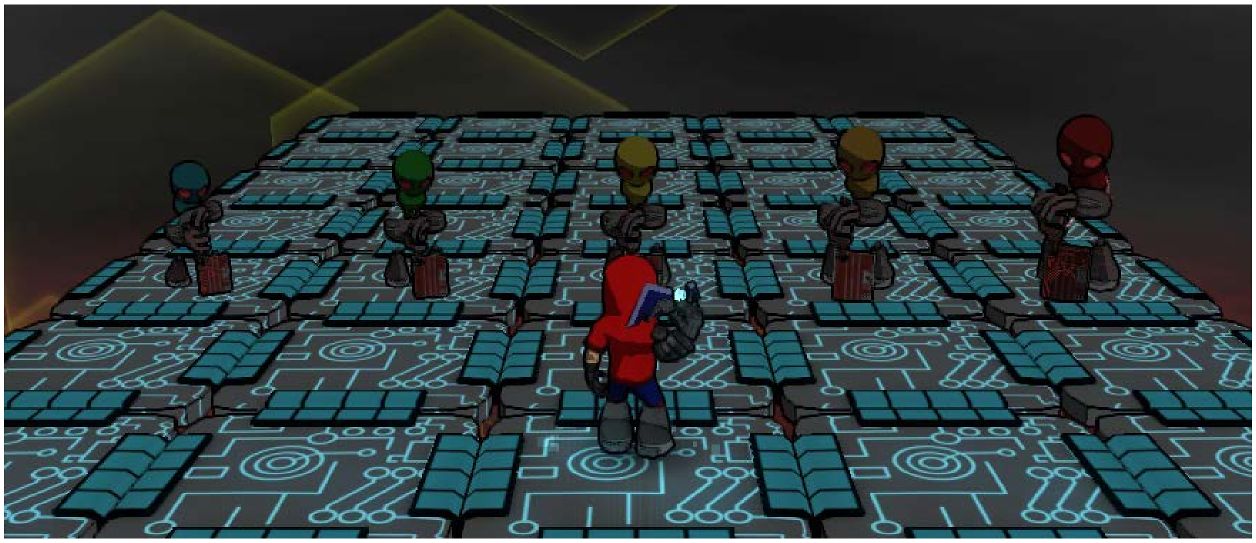


Figure 49: Variations on the same type of enemy in scale and color scheme

Enemies can vary in difficulty; two enemies of the same type may not have the same difficulty. Enemies that are more difficult are larger in size, and the color palette used for the enemy goes from cool colors to warm colors (blue, green, yellow, orange, red).

All enemies drop reward items when they are defeated, representing the "corrupted data" they are releasing. Most enemies drop **Health Data** that immediately heal the player and **BitBucks** that the player can use to buy upgrades at **Vending Machines**.

Spider Virus

These are small enemies that are fairly easy to defeat alone, but can replicate frequently. They are susceptible to most attacks, and the player is free to use any preferred strategy to dispatch them, though it is advisable to destroy them quickly.

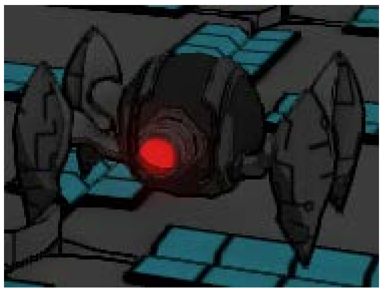


Figure 50: A spider virus

Spider viruses are designed to be multi-purpose, with both long-range and close-range attacks. In a large arena, groups of spiders can be deadly when targeting the player from far away, but their attacks can be avoided by jumping or dodging, and all of their attacks can be countered.

Soldier Virus

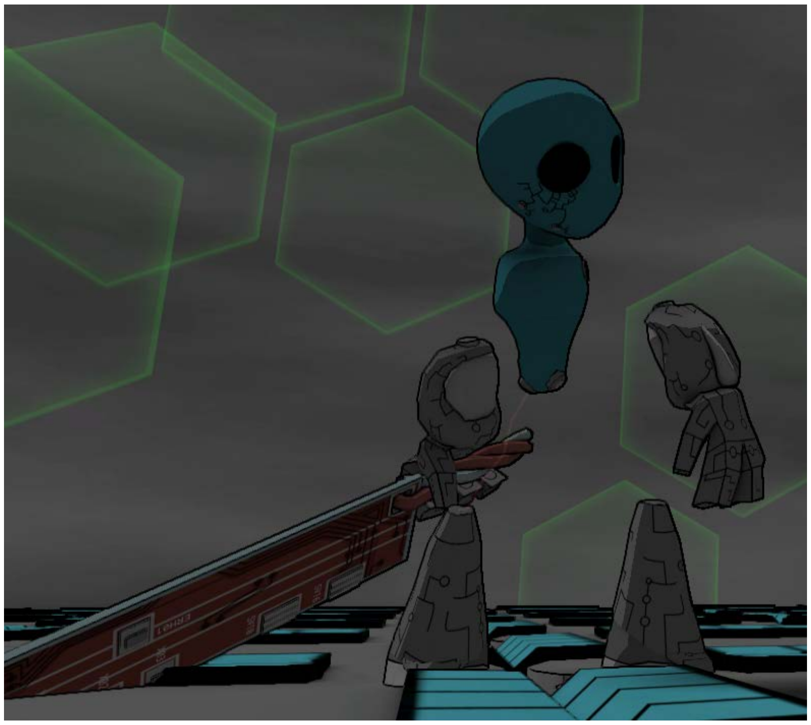


Figure 51: A soldier virus

These viruses are larger and tougher than their spider counterparts, but replicate less frequently. They lack a long-range attack, but their close-range attacks have higher damage and a wide range, which makes jumping a less viable avoidance strategy and encourages the player to use their blocking or dodging ability to mitigate damage. These enemies are also more likely to block, dodge, or retaliate player attacks.

The player benefits by putting distance between themselves and these enemies, and by making liberal use of blocking and dodging.

3.7.13.Reward Objects

Mid-way through designing and developing *Xeero*, it became clear that the player needed more incentives to perform the tasks the game was asking of them. This observation led to the development of reward objects.

Two primary reward objects were implemented: **Upgrade Chips** (discussed previously) and **BitBucks**, together with the objects used to facilitate them.

Most of the rewards are entirely optional for the player. Small rewards are easy to obtain, while larger rewards require extra challenges for the player to complete.

BitBucks

BitBucks are the currency of Xeero, and the basis of the game's reward system. They are dropped by enemies when they are defeated, but can also be scattered around levels for the player to collect. The location of the currency varies, with low value money being easily reached, but higher, more valuable denominations farther away or requiring more finesse or puzzle-solving. BitBucks are designed to be very frequently picked up, giving a small, but consistent reward to the player for progressing. They can be used to

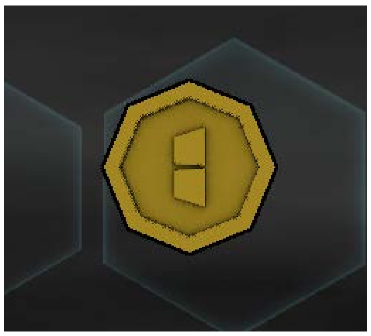


Figure 52: A single BitBuck

entice the player to move to one area or another or to point them to an important object for them to focus on.

Health Data

These rewards immediately restore some of the player's health. They offer respite to the player between particularly dangerous encounters in which the player may have taken a large amount of damage. Health data can be scattered sparsely through levels, and can also be dropped by defeated enemies.

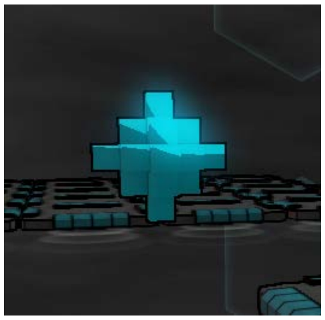


Figure 53: A piece of health data

Archives

Archives contain both **BitBucks** and **Health Data** in small quantities, and are refreshed each time a level begins. The player can "attack" them to collect the rewards concealed within. These can be placed in combat areas to give the player a source for more health if they run low during the encounter.

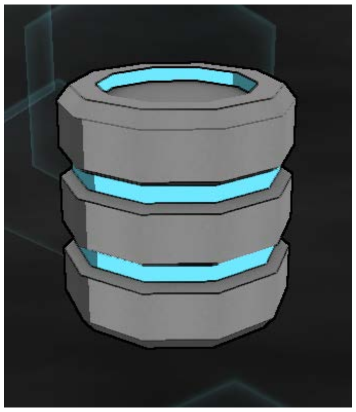


Figure 54: An archive, containing compressed health data and BitBucks

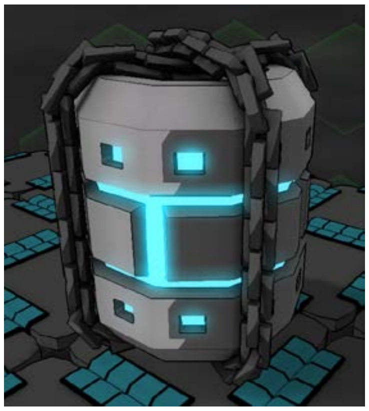


Figure 55: A big archive, containing a large amount of BitBucks and health

Big Archives

These one-time rewards contain large quantities of **Health Data** and **BitBucks**. They are placed in optional locations that require substantial effort to reach.

Vending Machines

Vending Machines allow players to make use of any BitBucks collected. They contain **Upgrade Chips** which the player can purchase and subsequently digitize. The upgrades gained through vending machines are entirely optional, but make some aspects of the game easier for the player.

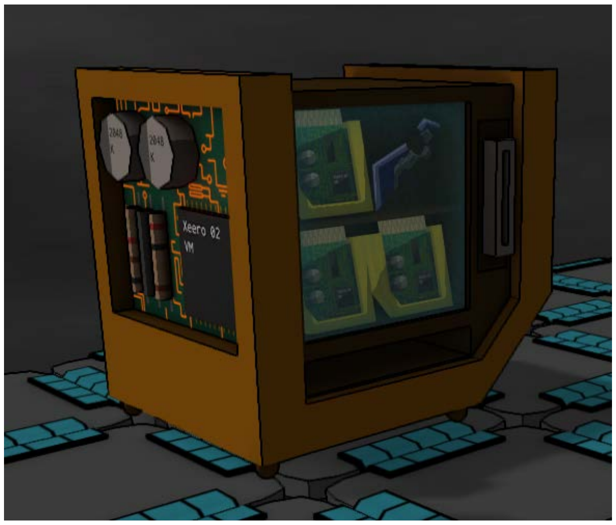


Figure 56: An idle vending machine

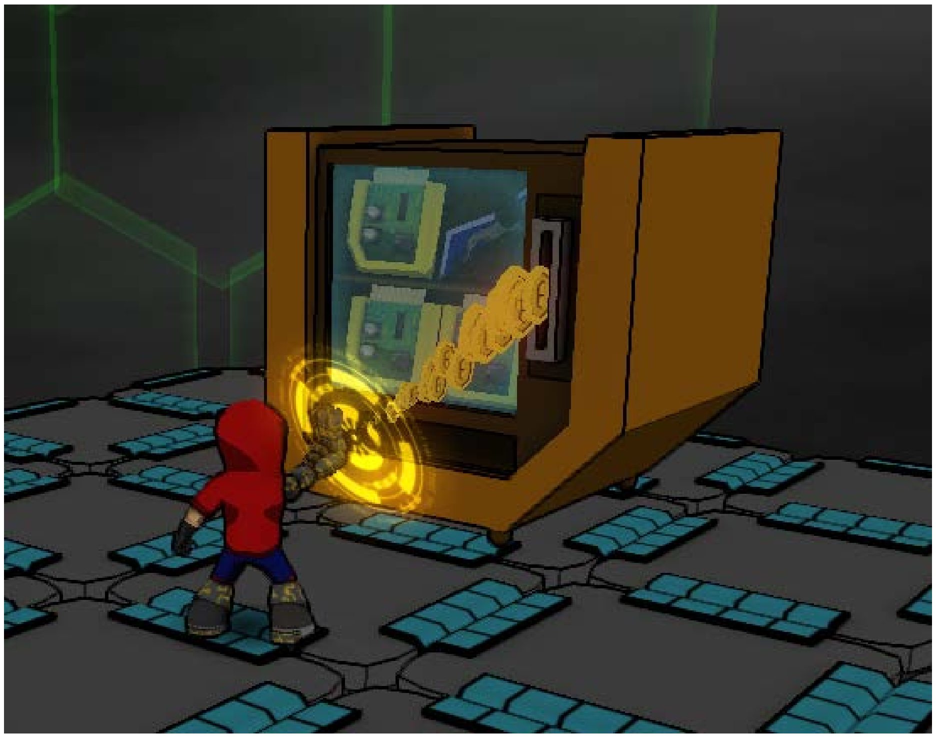


Figure 57: Xeero feeding money into the vending machine

3.8. HUD

The following section describes the elements of the heads-up display (HUD) used in *Xeero*.

The HUD is where most of the information needed by the player is located. Unlike some action games, but following the conventions of platformers, *Xeero* attempts to keep the full state of the player on screen at all times.

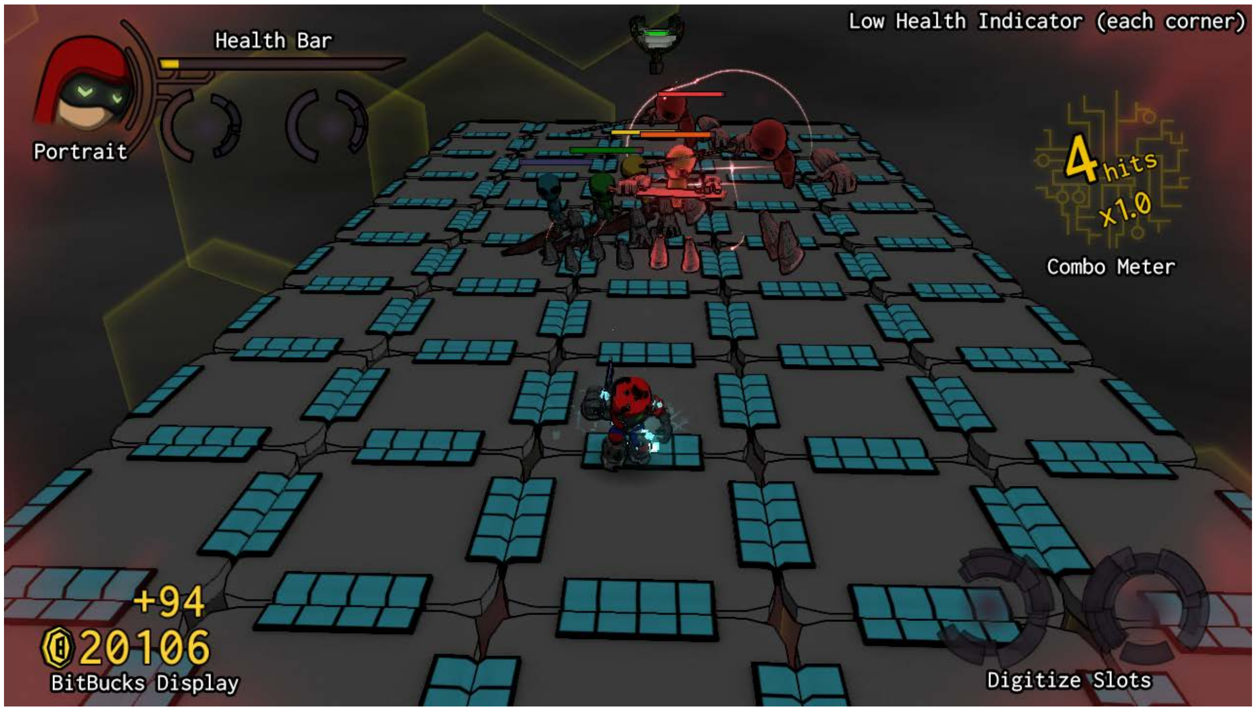


Figure 58: The HUD of Xeero*, marking relevant items*

3.8.1. Health Bar

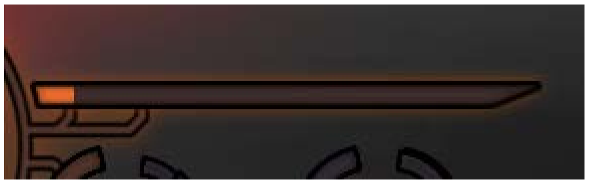


Figure 59: The health bar when Xeero has low health

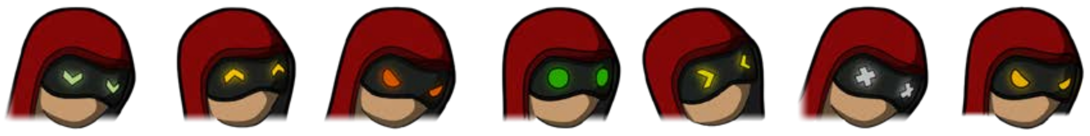
The health bar, or "data fidelity meter," indicates the player's health. As Xeero takes damage from attacks, his data becomes more corrupted, and he loses "data fidelity." When the bar is empty, Xeero "crashes," and is sent back to the last hit breakpoint to recompile.

The health bar ticks down as the player takes damage, leaving behind a red indicator to show the player how much health was lost. As the player loses health, the bar changes color from blue to red and begins to flash when the player has low health.

3.8.2. Low Health Indicator

When the player has low health, the edges of the screen glow red and a quiet, but noticeable tone begins to play, warning the player.

3.8.3. Character Portrait



The character portrait shows the emotional state of Xeero over the course of the level. Xeero's portrait will emote contextually as events happen in the game. Most often, Xeero is idle. He will begin to glare when he enters combat, and he will have a more intense glare as he

Figure 60: Some of the different states of the character portrait

performs attacks. He squeezes his eyes shut when hit in combat, and his head will droop when he has low health. His head droops more intensely when he "crashes." His eyes will be upturned happily when he collects **Health Data** or **BitBucks.**

3.8.4. Digitize Slots

The digitize slots show the player what is currently in their gauntlet. The player is allowed to store up to two objects in the gauntlet (one left and one right), reflected by the icons in the HUD.

The indicator is designed to display information for each state the object is in within the gauntlet and to have a visual cue for every action the player can perform with the object. The indicator matches the state of the object, animating to show it being digitized to or materialized from Xeero's gauntlet. It shows the object crumbling when it has been disassembled, and wipes it away when the object has been purged by a **Purge Gate**.

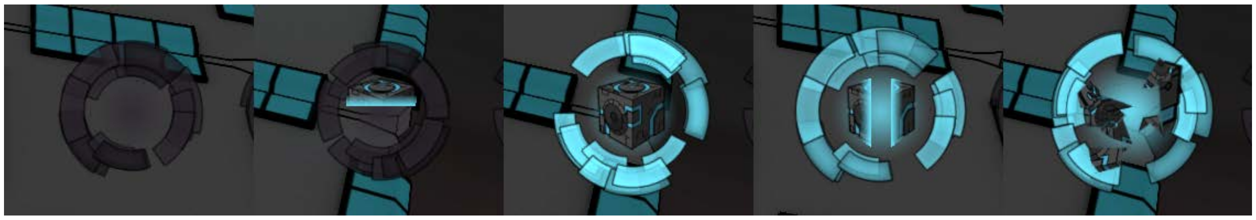


Figure 61: Some of the different states of a digitize slot

When the slot is empty, the player can hold the digitize button to aim at the object they want to digitize. The HUD will show a ghost of the object that *will* be digitized in the appropriate slot. When the slot is filled and the materialize button is held, the object will glow in the appropriate slot to show which object will be materialized.

3.8.5. Combo Meter

The combo meter is designed to add a sense of satisfaction and reward to combat (see **Audience and Experience Goals**) and to inform the player on helpful strategies in combat.

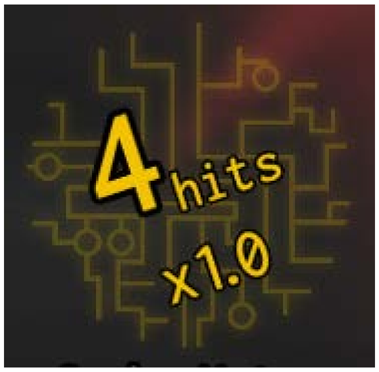


Figure 62: The combo counter and current multiplier

When the player successfully hits an enemy (or successfully blocks and counters an attack - see **Mechanics - Defensive Abilities**), the combo meter will increment for every object hit. Upon reaching predetermined threshold values, the player will gain a multiplier on the hit counter. The multiplier increases the amount of **BitBucks** and **Health Data** that is dropped by **Enemies** and **Turrets** when they are defeated. The player can up to double the amount of money and health gained from combat. Whenever the player is hit and stunned by an attack, the counter will tick snap down to the next lowest multiplier value.

The player doesn’t have to maintain a single chain of attacks in a combo to keep ticking the hit counter, and the counter will maintain its value while the player is out of combat.

The hit counter is designed to add a sense of reward to combat; the player should feel satisfaction for building up a high value in the hit counter. Each successive hit causes a short animation to play on the counter, where the number will grow and shake. The intensity of the animation increases with the number of hits on the counter, and another, more intense animation plays whenever the multiplier increases. The intensity of the animations is designed to bolster the visceral impact of connecting a strike in combat.

Additionally, the inclusion of the hit counter helps encourage the player to adopt a strategy in combat. If a player chooses to focus on building up a higher hit counter, the player is then encouraged to avoid getting struck in combat, and the player may then try to use their defensive abilities more often.

3.8.6. BitBucks Display

The BitBucks display shows the player how many **BitBucks** they currently have in their possession. The display uses animations to bolster the reward sensation from collecting **BitBucks** (see **Objects - Reward Objects**).

When the player picks up BitBucks, they is added to a running total in their wallet. After a beat, the money gained will rapidly tick down, flowing into the player's total wallet. When the running total reaches zero, the player's total will pulse and a "cha-ching!" sound effect plays. The intention is for the player to feel the relative weight of adding one **BitBuck**, ten, or a thousand.

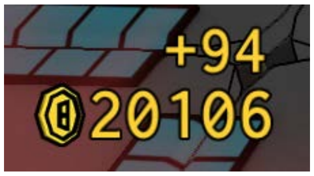


Figure 63: Gaining money

When the player purchases an **Upgrade Chip** from a **Vending Machine**, the money ticks down with a similar animation as the Xeero feeds money into the machine.

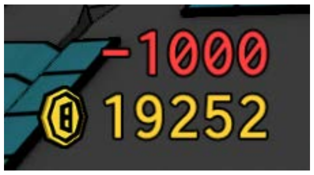


Figure 64: Losing money

3.8.7. Enemy Health Bars

Enemy health bars show the current health of enemies the player is fighting, similar to the player health bar. The bar appears over the head of the enemy, but only when the enemy is close to the player and is taking damage.

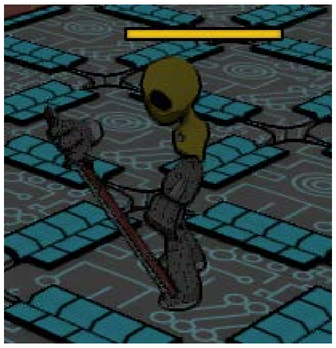


Figure 65: A health bar over the head of an enemy

3.9. Level Editor

The following section describes the level editor created for *Xeero*, motivations and design decisions behind its construction, and how it is used to construct levels in the game.

3.9.1. Motivations

Early in development, a flexible format was created to allow levels to be developed programmatically. Originally, the intention was to allow levels to be mocked up on paper before being entered into a text file to be read by the game.

However, levels quickly grew in complexity, requiring hours of manual entry to create. Because the game had to be loaded before the results of the entry could be viewed, much of the design process to create a level matching a drawn mockup involved writing to a file, opening the game, closing the game, tweaking the level file, and opening the game again. Additionally, when the levels were play tested and tweaks needed to be made, the specific section of the level had to be tracked down by matching coordinates to the visual layout of the level. Using manual entry text files made both designing and implementing levels a laborious task.

From the very first prototypes of the game, *Xeero* was designed to use modular levels: each level is made up of small pieces common to most levels (for more information, see **Appendix A**, which describes how space is partitioned for level building and player navigation).

Levels in *Xeero* avoid large set pieces or long stretches of custom content that can only be used for a single level. Reusing as much content as possible was a top design priority.

3.9.2. Programmer

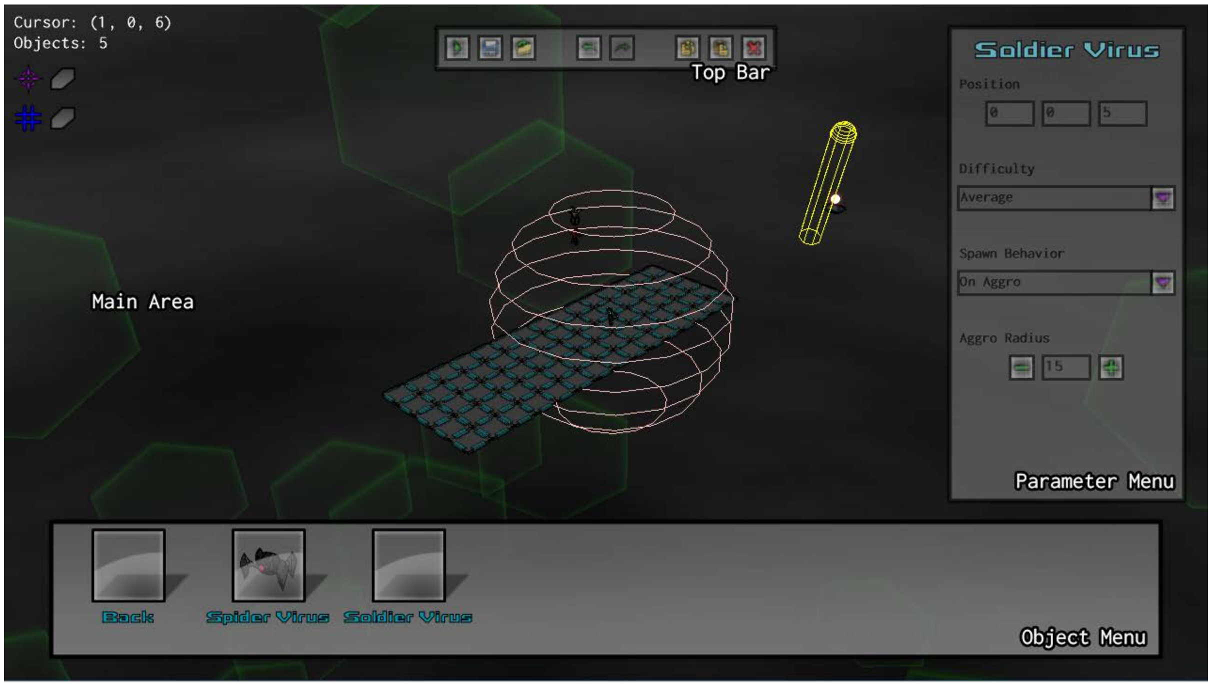


Figure 66: The interface of "Programmer," with relevant items marked

Together, these motivations informed the creation of the level editor, Programmer. This tool allows the modular level elements of the game to be dragged and dropped onto a level, allowing them to be quickly mocked up, tested, and changed.

Every object in the game can be placed into levels from the menu in the UI marked **Object Menu**. The designer can drag and drop the object from their pane in the menu or can double click to add the object to the level. Once the object is in place, various parameters can be set on the object (like how long **Falling Platforms** take to fall or which upgrade the player gains from an **Upgrade Chip**) by using the menu marked **Parameter Menu**. Chunks of the level can be selected and moved by clicking and dragging on either a single object or on a selected group of objects by dragging on empty space in the main area marked **Main Area**. The upper menu marked **Top Bar** allows the designer to copy and paste sections of the level, undo and redo actions, and save and open different levels. The upper menu also allows the designer to play the level as it is being constructed, allowing for rapid testing and refinement.

The levels created with the level editor can be saved to and loaded from human- and machine-readable XML.

3.9.3. Linking

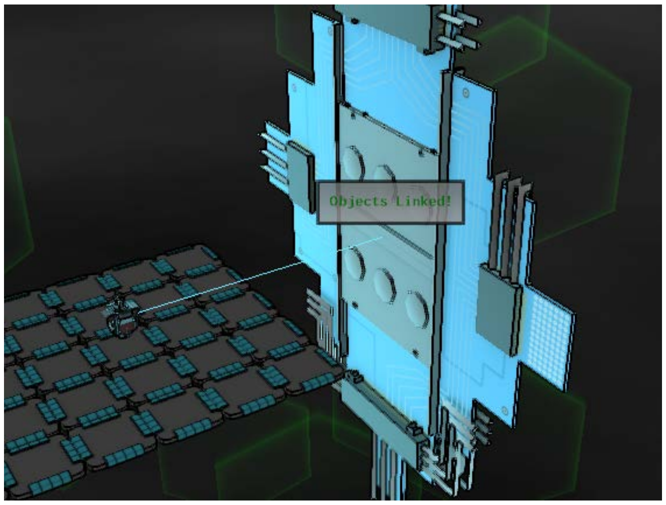


Figure 67: Linking a switch to a gate

Some objects can define relationships with other objects in the game. Such “links” are defined by right-clicking and dragging between two compatible objects. In *Xeero*, most link relationships are generalized into “switches” and “receivers.” Any “switch” object (like **Buttons, Hit Switches,** and even objects like **Turrets**) can be linked to any “receiver” object (like **Gates, or** any hazard that can be enabled or disabled), allowing the switch to activate or deactivate the receiver in the level once the switch has been activated.

3.9.4. Mockup Objects

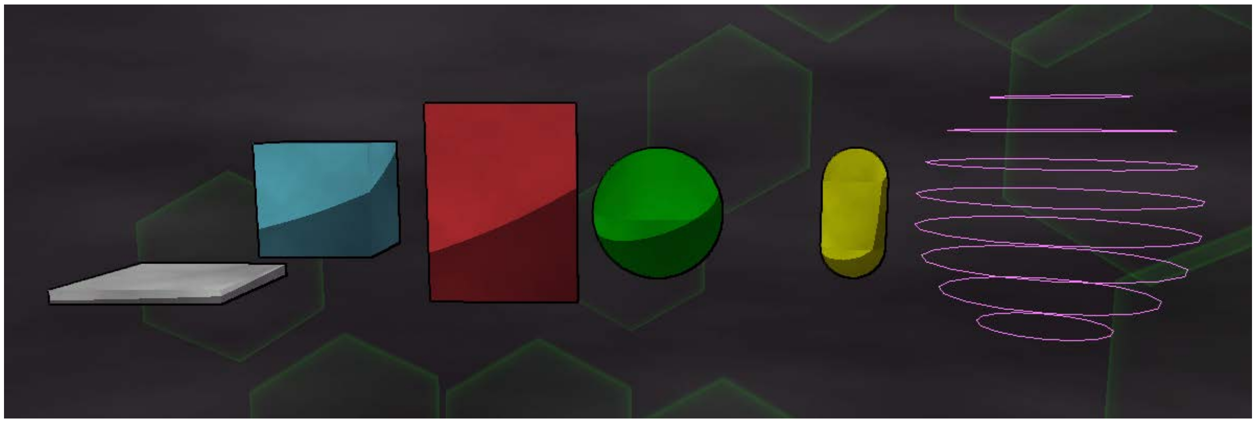


Figure 68: The mockup objects in different colors

Additionally, the editor gives the designer access to “mockup objects,” blank primitives of different shapes -- such as boxes, spheres, and capsules -- that can be used to quickly sketch out levels. These objects can be color coded by the designer, and the designer can type in notes that are stored with the object to distinguish their function.

Mockup objects can be used to sketch out the proportions and platforming challenges of levels without needing to set parameters for every object. These can also be used to create the structure of a level for which the required objects haven’t yet been created. The figure below shows the first hub level mocked up and fully realized.

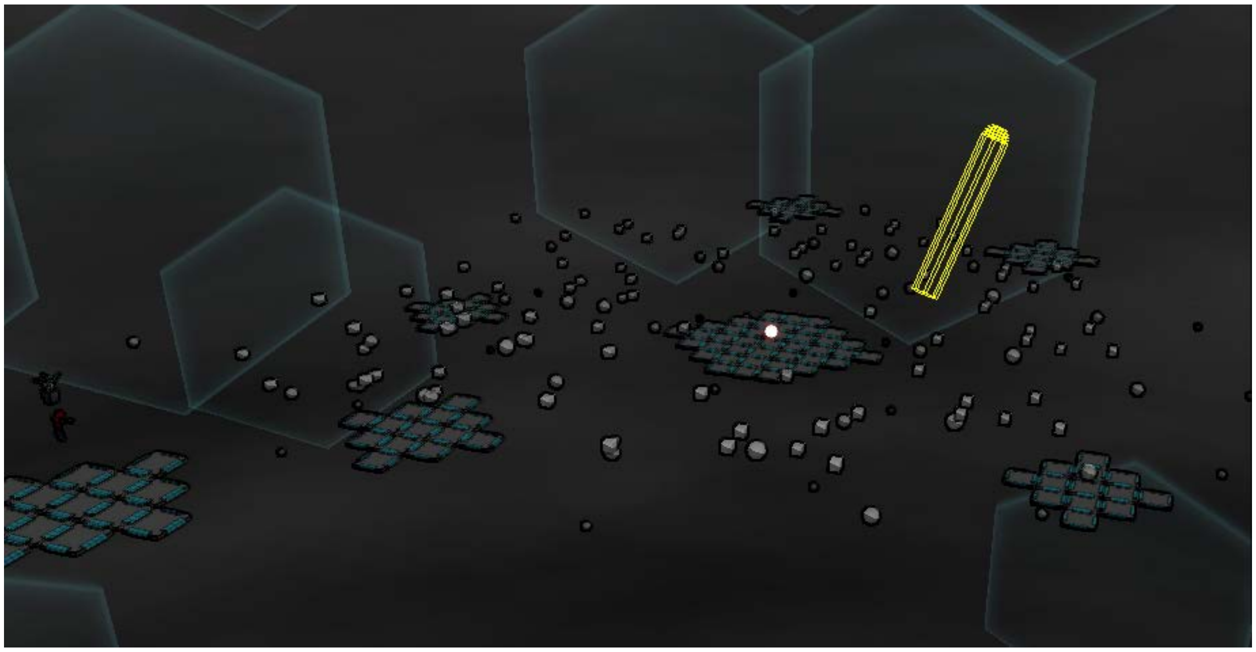


Figure 69: A version of Hub 1 using mockup objects

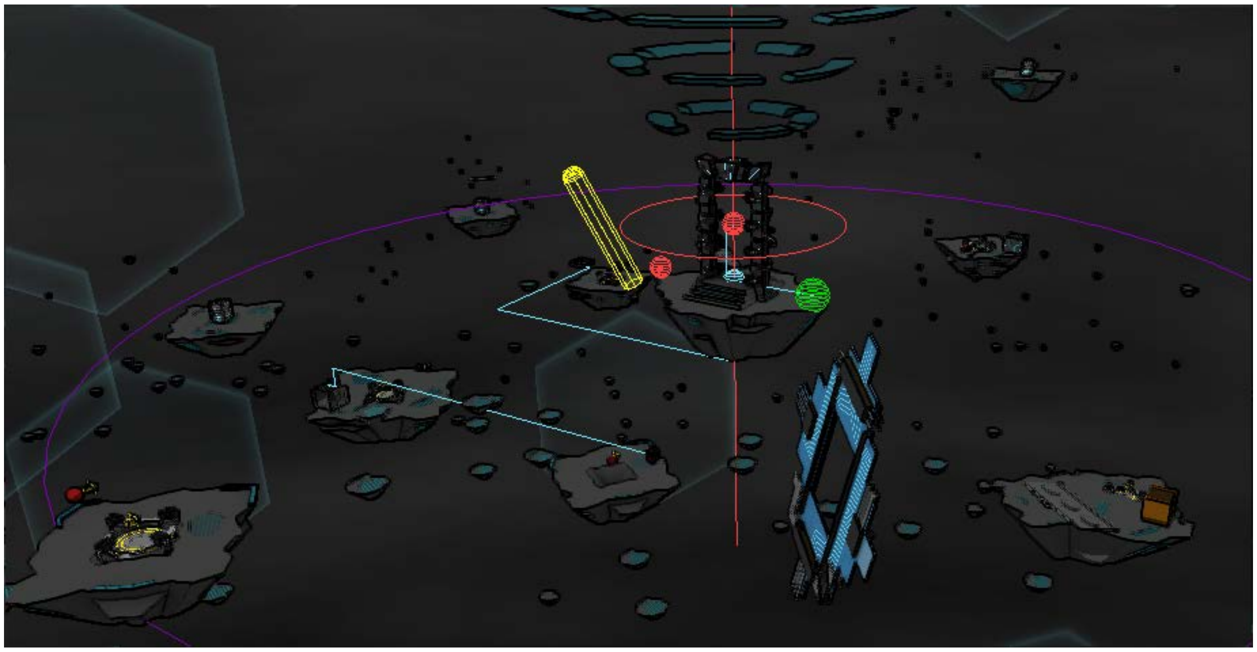


Figure 70: Finalized version of Hub 1

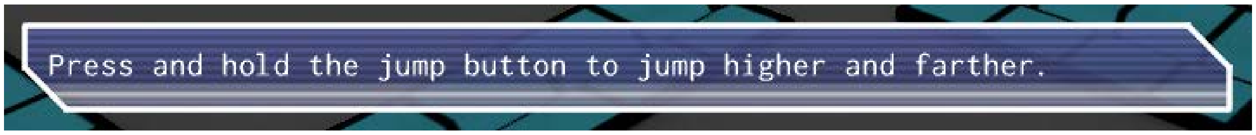
3.9.5. Level Sharing

Additionally, the level editor is built into the executable game as a game state that can be triggered from within the game. As a result, the level editor can construct levels to be saved and used for players of the game, but also allows players to create, save, edit, and transmit level files. The level editor, from its conceptualization, was design to be packaged with the game, allowing players to use the simple drag-and-drop controls to construct and share their own levels.

3.10. Tutorials and Help

As discussed in the **Inspirations** sections, *Xeero* drew from games like *Portal* for constructing tutorials for the player. Many puzzles of *Xeero* are designed to teach the player about different controls and the properties of various objects.

The tutorials for *Xeero* went through several iterations over the course of this project, each time changing based on feedback and observations from players. Initially, the only method for teaching the player new information was through "tutorial tooltips (pictured below)."



Press and hold the jump button to jump higher and farther.

Figure 71: A sample tutorial tooltip

These tooltips displayed all of the information the player was given about the game, from controls, to mechanics, to object properties. Inside levels, there were orbs the player could touch, which would trigger the tooltips to appear. The intention was for the player to see and touch each orb, read the information, and use that information to proceed in the level. However, initial playtest showed players avoiding the orbs, missing the displayed tooltip, or not reading the messages they contained. Then, inevitably, the player would attempt -- and fail -- the section of the level the tutorial was attempting to teach them about, and the player would not understand what they were intended to do.

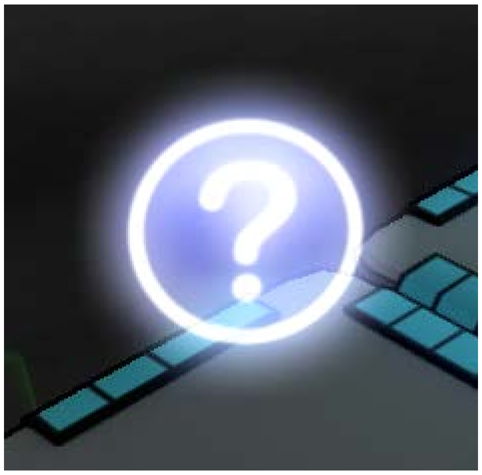


Figure 72: A tutorial bubble

To remedy these issues, four new types of tutorials were introduced to *Xeero*: **tutorial "equations," info pages, revamped tutorial tooltips, and loading tips**.

3.10.1. Tutorial Equations

Initially, information about the controls of the game (e.g. "press A to jump") was given exclusively through tutorial messages at the bottom of the screen. This presented two problems. Messages quickly became verbose for simple mechanics, and, in areas where there were several control instructions in succession, they gave the players more text than they were willing to read. Additionally, the text of tutorial messages was defined within the level editor, making all of the text fixed within the level. However, as discussed in the **Controls** section, all of the mapped buttons could be remapped by the player. Further, the game allows the player to use either the mouse and keyboard or a gamepad when playing on the PC. Taking all of these factors into account created awkward tutorial messages such as. "Press 'A' or space/left control to jump (these controls can be remapped in the options menu)."

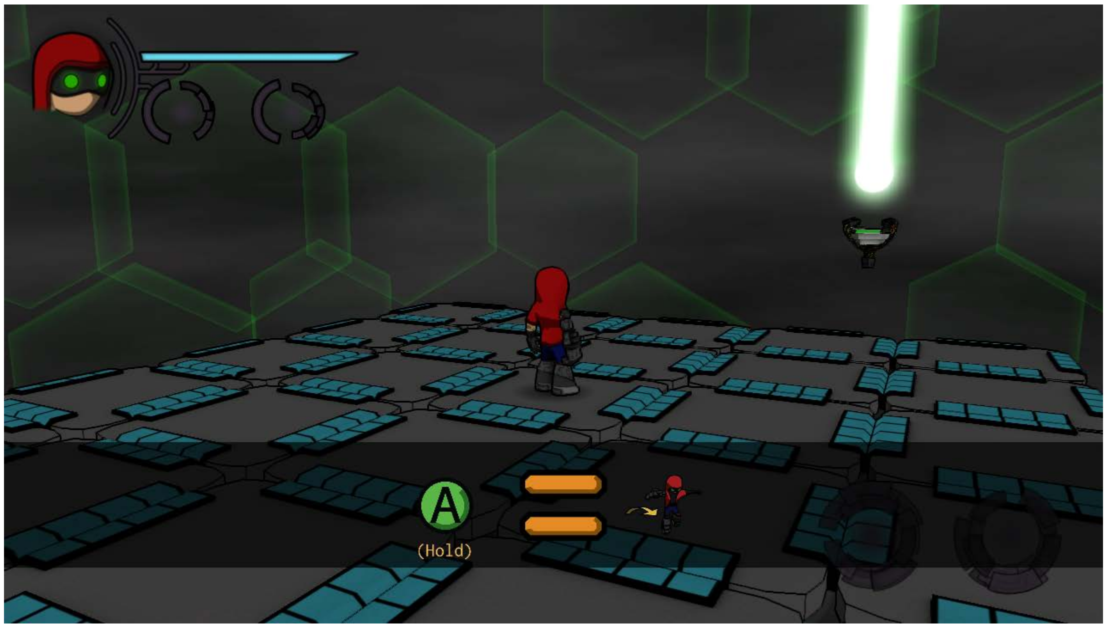


Figure 73: A tutorial equation as shown to the player

Visual tutorials sought to remedy these issues. At certain sections of the game, a tutorial (like the one picture above) would appear near the bottom of the screen. These pictorially depicted controls for the player to use to aid the player quickly learning new information. The player would see a large image for a button and a symbol and would be encouraged to try the button and experiment.

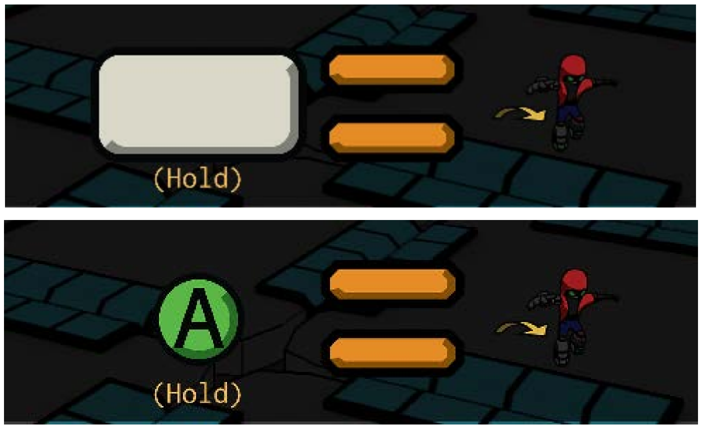


Figure 74: A tutorial equation for mouse/keyboard and gamepad controls

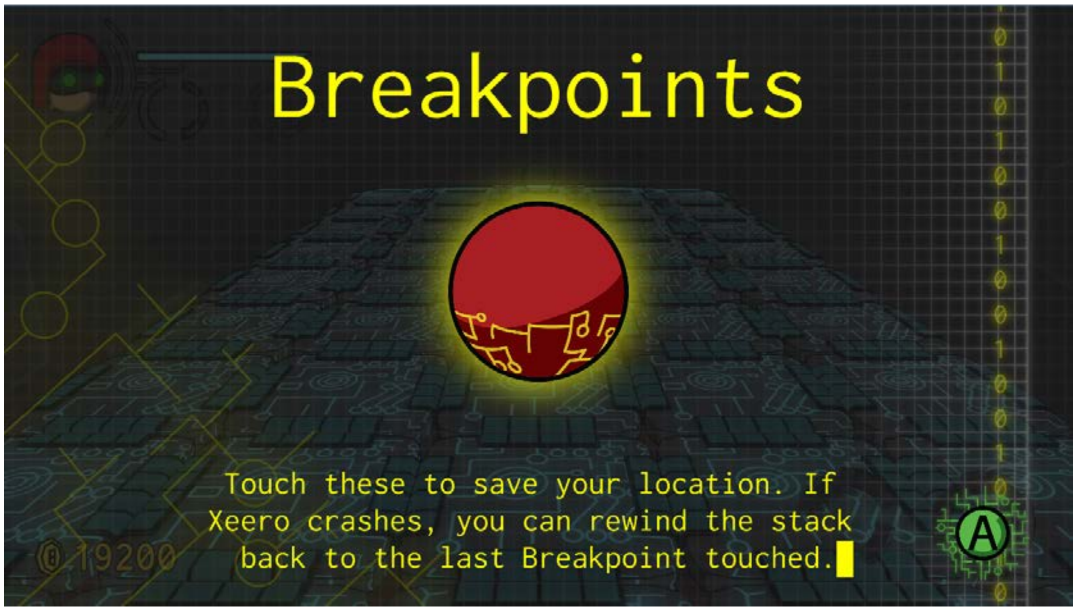
Additionally, the visual tutorial system recognizes the possible control configurations the player can use. Each button on the gamepad, keyboard, and mouse has an icon, and the equation can show whichever button is mapped to that control. The tutorials also can detect which hardware the player is using to play the game (mouse/keyboard or gamepad), and will show the appropriate buttons to the player.

Through pictorial display and dynamic button icons, visual tutorial "equations" reduce the verbosity and inaccuracy of early tutorial messages.

3.10.2. Info Pages

While visual tutorials are designed to teach the player about controls used in the game, Info Pages are designed to teach the player about specific objects they can encounter in levels.

Some information about objects is critical to the player's understanding of the mechanics of the game. When this information was displayed in a tutorial message, the player could ignore or unintentionally miss the information presented to them.


Breakpoints

Touch these to save your location. If Xeero crashes, you can rewind the stack back to the last Breakpoint touched.

Figure 75: An info page for a breakpoint

The info pages are full-screen animations that display the name of the object, a large icon of the object, and a brief description. While the info page is running, the game underneath is paused, and the player must press a button to acknowledge the info page to dismiss it. This large animation immediately captures the eye of the player, and, while it can’t force the player to read anything, it at least ensures that the player is aware that there is information for them to learn.

When a player approaches an important object in a level for the first time, the info page is triggered after the player gets within a certain distance. Info pages can also be used for player upgrades. When the player gains a new ability (called a new “subroutine” in-game), the player is also shown an info page detailing the upgrade they have just received.

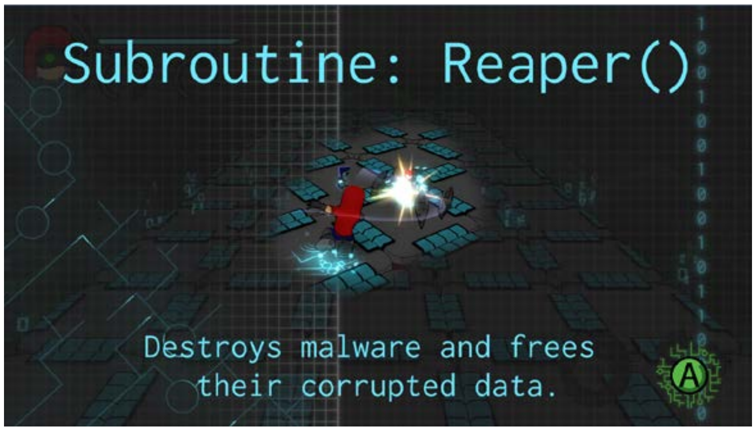


Figure 76: An info page for an upgrade

3.10.3.Revamped Tutorial Tooltips

Some information presented in the game is location-specific, not mapped to a specific object, details mechanics in a way that is difficult to present in pictures, or is not critical enough to block the player from playing a level. For this information, the original tutorial messages are used, but they have been modified to mitigate the issues they originally presented.

Initially, these tooltips could only be activated by the player choosing to touch a small “tutorial bubble” within a level. The activation method for these tooltips have been expanded with a series of objects classified as “conditionals.” Conditionals are objects that can be manipulated in the level editor that act like drag-and-drop Boolean logic. Conditionals can be linked to a tutorial message, so for however long the condition is true, the message will be displayed for the player.

There are a variety of different conditionals that can be hooked to tutorials. Most directly related to their original activation method is the “Touching” conditional, which is triggered when the player touches an invisible sphere of arbitrary radius. This allows a tutorial message to be activated when the player enters a section of the level, rather than by touching a small sphere. A "Died Here" conditional triggers when the player "dies" while touching the trigger. This is used to display messages to the player when they repeatedly fail a certain section of a level. Some conditionals can be very specific, like "Ledge Hang" conditionals that are triggered if the player is hanging off of any ledge or a "Materialize Blocked" conditional that is triggered whenever the player tries to materialize an object (see **Mechanics - Digitizing and Materializing**) when there is another object in the way.

These conditionals, like Boolean logic, can also be linked together. The level designer has access to an "And" and an "Or" conditional that can link two conditionals together using the appropriate logic.

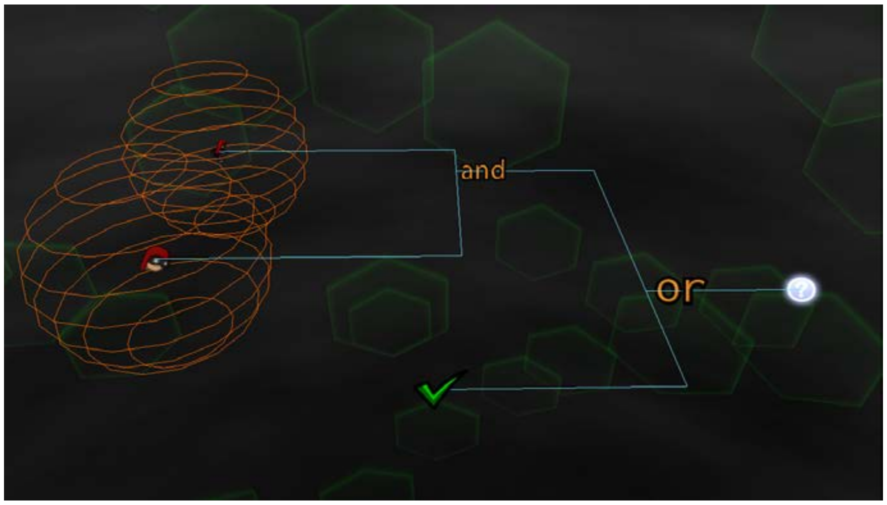


Figure 77: Linking conditionals with "and" and "or" conditionals

3.10.4.Loading Tips

Some types of information don't fit any of the above categories. Some information is miscellaneous, not tied to a specific location in a level, or is just informative about the world of the game. These short (usually one sentence) pieces of information are given to the player in the loading screens of the game that appear between levels.

This information can sometimes be dependent on the current progress the player has made in the game; the tip "you can only block attacks from the front" only applies after the player has received their weapon and has been taught to block attacks. Thus, each tip in the game can be triggered to show only after the player has beaten a specific level. The tips can also be set to hide themselves after a player has beaten a specific level.

Through pictorial tutorial equations, full-screen info pages, conditionally triggered tooltips, and miscellaneous loading screen tips, each type of instruction given to the player is dispensed in a way designed to most effectively teach the required information.

4. Visual Art

Eric was also responsible for the visual art of *Xeero*. The following section describes the visual style developed for the game, discusses some of the technology used, assets created, the motivation for particular styles, and the art that inspired the design of the assets.

4.1. Introduction

The initial direction for the concept and art style of the game was heavily influenced by the creation of the protagonist. Xeero was the first element of the game conceptualized and went through a number of iterations.

In the first iteration of Xeero, the computer motif was not established; the character could have existed in a different fantasy or sci-fi world. Despite this, the first iteration of Xeero maintained many of the features that would carry over to his final version: he retained goggles, gauntlet arm, and large boots, albeit in a different style.

The next iterations of Xeero softened the edges on the character, attempting to make the character seem less potentially malicious and friendlier. The figure to the right shows the evolution as the proportions of the character were finalized.

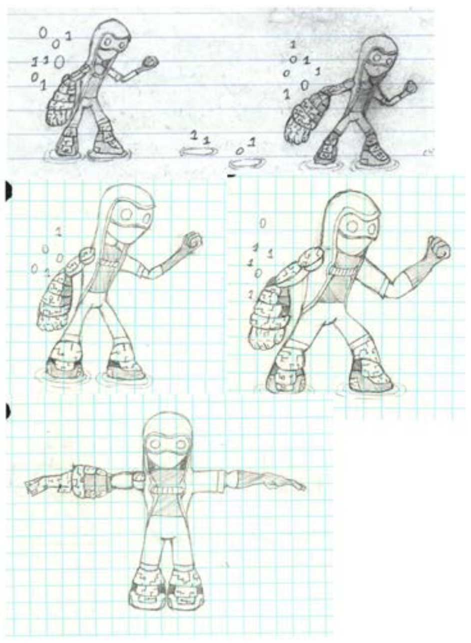


Figure 78: Early sketches if Xeero

After the character concept was designed, Eric next designed the concepts for the types of objects Xeero would use, and the concept of “digitizing” and the computer motif was developed. The figure below shows some sketches of the initial objects designed for the

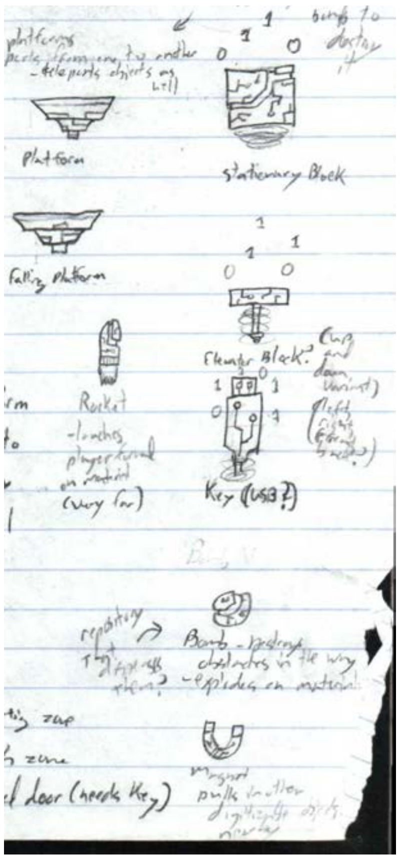


Figure 79: Early object sketches

game, including a block, an elevator platform that would raise and lower for Xeero to stand on, a key object that unlock locked doors, and bombs that would destroy certain obstacles.

As the protagonist and associated objects took shape, Eric began designing the aesthetic the rest of the game would employ.

4.2.Style - 3D

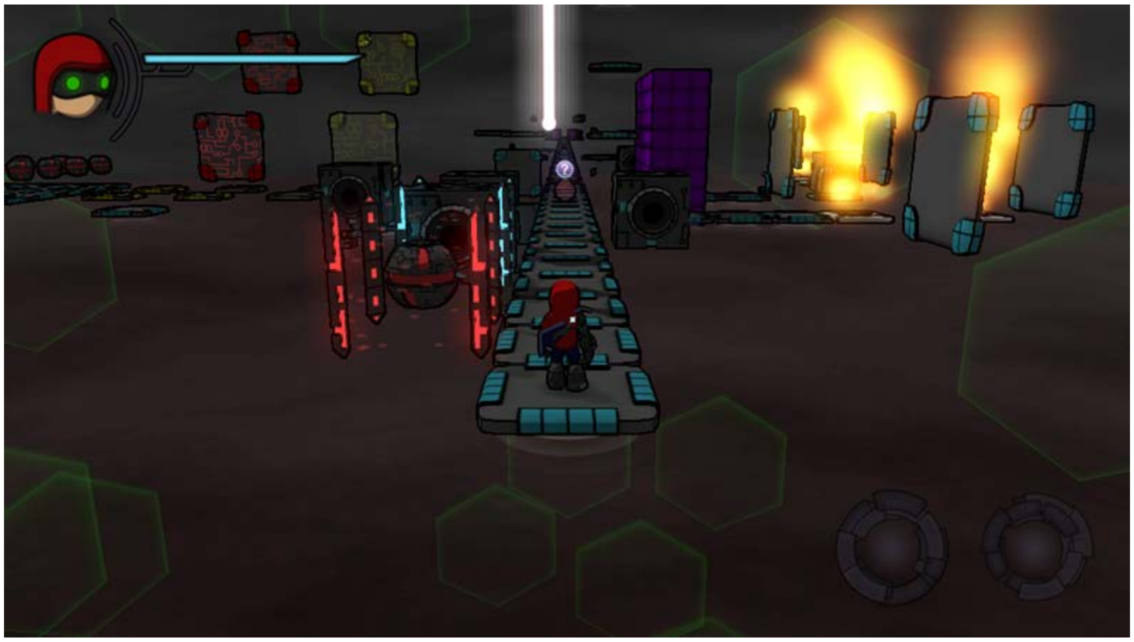


Figure 80: A sample level in Xeero

From the start of the project, Eric had a preference for stylized 3D rendering, stemming from an early interest in animated movies. The creation of the art for the game was focused on simulating “cartoon-like” visuals and animations. *Xeero* also drew inspirations from video games using a similar style.

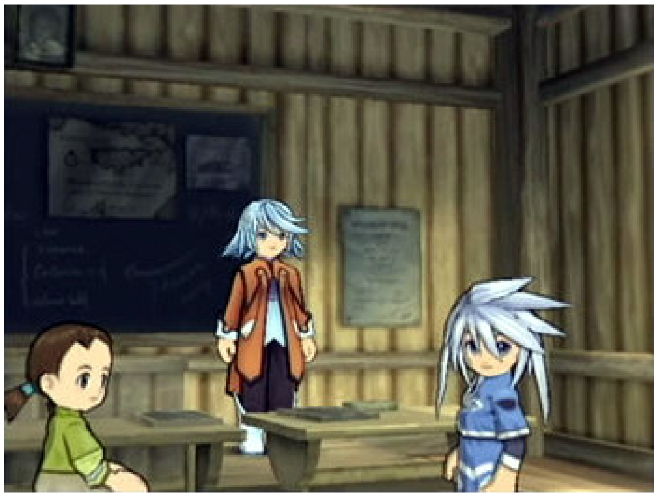


Figure 81: Colors and character proportions in Tales of Symphonia (Acev)

Xeero was inspired by games of the “Tales” series. Games in these series have heavily anime-inspired, stylized graphics. Character proportions are often shorter, and the colors are often solid and bright. Partially a product of the technical limitations of their time and features of anime, often models contained simple shapes and colors, with much of the detail of characters and props informed by shading. In several 3D games of the series (like Namco’s *Tales of Symphonia* pictured above), the games use a combination of cel-shading (using flat, binary “light and dark areas” for shadows) and pre-baked smooth shading on the textures of models. Additionally, to help distinguish the objects from the environment and mimic the drawing techniques used in cartoons and anime, characters and props all have black outlines.

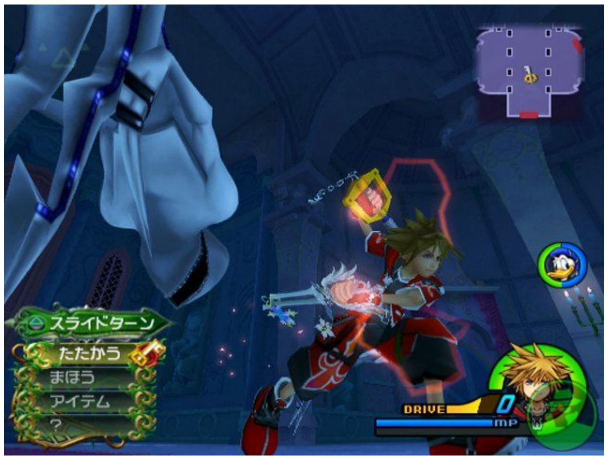


Figure 82: Texture baked shading in Kingdom Hearts II (Renmiri)

Games in the *Kingdom Hearts* series also provided inspiration for art. While the early games do not use cel-shading -- or dynamically generated shading of any kind -- the games still attempt to create visuals reminiscent of the style of Disney animations. There is an emphasis on large proportions for both characters and props, and objects have soft shading baked into the textures of the objects. Original character models are created to bridge the gap between the large disproportions used in Western animation and the more realistic proportions used in anime.

Kingdom Hearts games also use an abundance of particle effects and elaborate character animations in combat. In line with the "moments" of gameplay *Xeero* attempts to evoke (see **Experience Goal and Audience**), *Kingdom Hearts* games give the player elaborate, "flashy and vicious" attacks the player can execute, using a variety of additional particle effects and acrobatic animations.

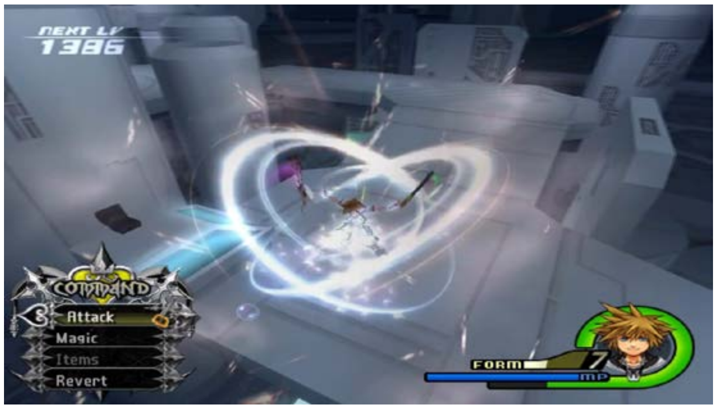


Figure 83: Use of particle effects in Kingdom Hearts II (WastedMeerkat)

Xeero also drew inspiration from *Bastion*. The worlds in *Bastion* are heavily modular, each level building itself around the player as the player progresses through it. This influenced the "modular" design of levels discussed in **Level Structure**.

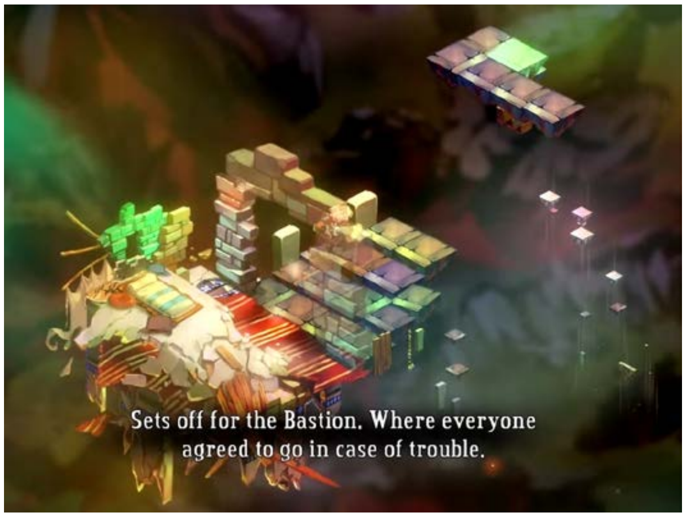

Sets off for the Bastion. Where everyone
agreed to go in case of trouble.

Figure 84: A level constructing itself in Bastion (Northernlion)

Xeero also drew from games like *Bastion* and games in the *Mario* series when determining the proportions and scale for the player character. In games from either series, based on their relative proportions to the environments in which they find themselves and the proportion of their body parts, protagonist are short, roughly half of the size of an average adult.

From these inspirations, the art style for *Xeero* was developed. The following sections will describe how these inspirations informed different aspects of the asset creation process.

4.3. Modeling

Like games in the *Tales* and *Kingdom Hearts* series, for both character and props, models were created with large, simple shapes and large proportions. For most objects, very few intricate details were developed, instead focusing on the general form and the silhouettes of the objects. Very little sculpting was used for props; most objects in the game represent some sort of "computer component" were composed of mostly hard edges. Normal maps are not used in *Xeero*, to intentionally keep the art minimalistic, to give as much visual information as possible in larger shapes, and to reduce the urge to continue adding finer and finer details to models as development continued.

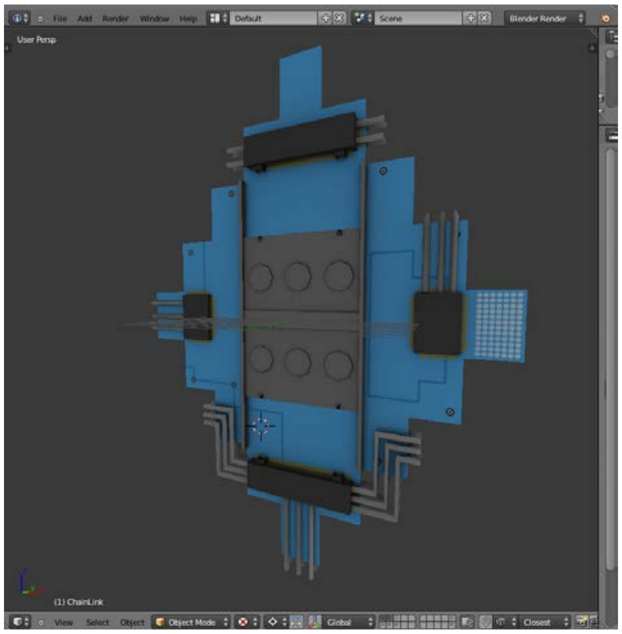


Figure 85: Large, simple shapes in models for Xeero

4.4. Color Schemes

Inspired from both games in the *Tales* series and the *Tron* series and independent artists, *Xeero* uses splashes of saturated, bright colors against mostly black and gray environments. Early in development, a "core set" of colors was developed; most of the predominant colors used in texturing and in particle effects are slight modifications in saturation and brightness of these core colors.

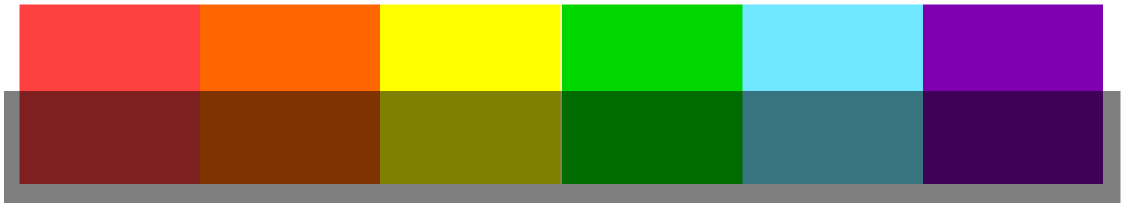


Figure 86: The fundamental hues used in Xeero

In texturing, most surfaces use a single, flat color based on the color scheme shown above. Then, like in early *Kingdom Hearts* games, a layer of soft shadows is baked into the texture.

Additionally, to emphasize the contrast between these vibrant colors and the black or gray environments, *Xeero* uses a full-screen bloom effect on brighter colors. This helps vibrant colors and bright particle effects show their intensity compared to the darkness of the rest of the world of *Xeero*.

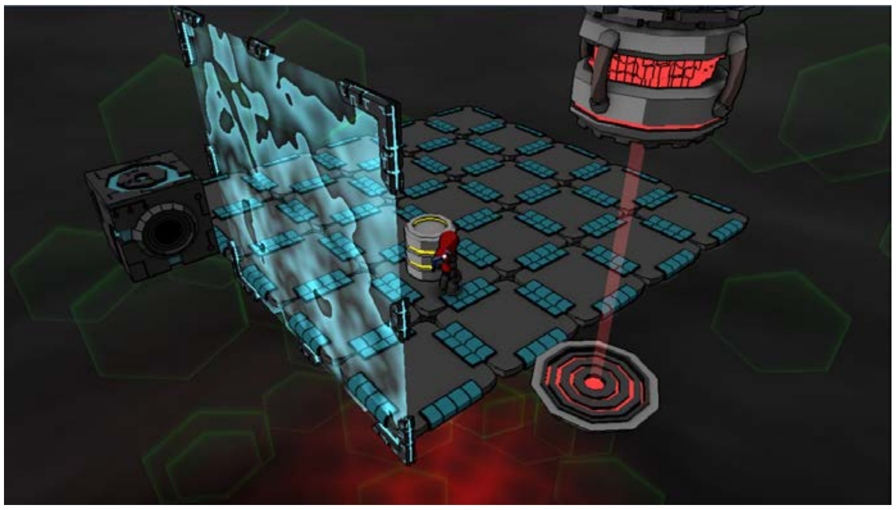


Figure 87: Full-screen bloom effect disabled

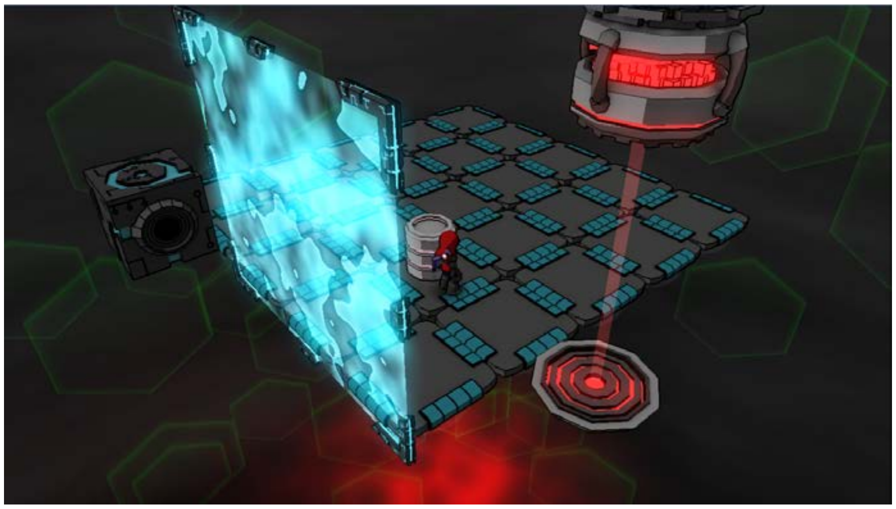


Figure 88: Full-screen bloom effect enabled

4.5. Lighting and Shading

While *Xeero* employs soft shadows baked into textures, the game also dynamically shades objects using cel-shading techniques similar to those in the *Tales* series.

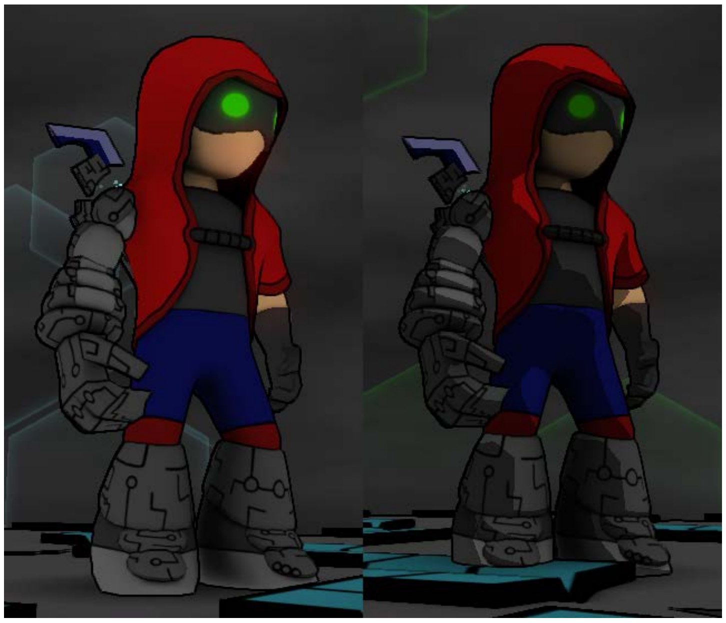


Figure 89: Xeero with and without cel-shading

In cel-shading, for any surface, light values are calculated by determining the angle the surface makes with the light source, like typical diffuse lighting. However, instead of allowing intermediate values of light and dark based on the angle of incident light, angles below a certain threshold are considered "light" and others "dark." Using this binary measurement, the surface is then shaded a preset shade based on whether it is "light" or "dark." Most levels of *Xeero* contain a single, directional "cel light," which is used to determine the direction and intensity of the cel-shading effect.

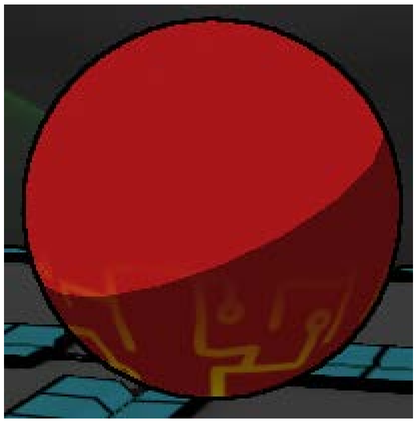


Figure 90: Shading in Xeero *has no gradients, only binary "light" and "dark"*

However, in addition to cel-shading, *Xeero* also uses more realistic lighting effects. For some visual effects that represent the release of "energy," like digitizing, fire effects, and explosions, colored lights are used. These lights use gradients calculated from the angle of the incident light ray and are additively added to the colors generated from cel-shading.

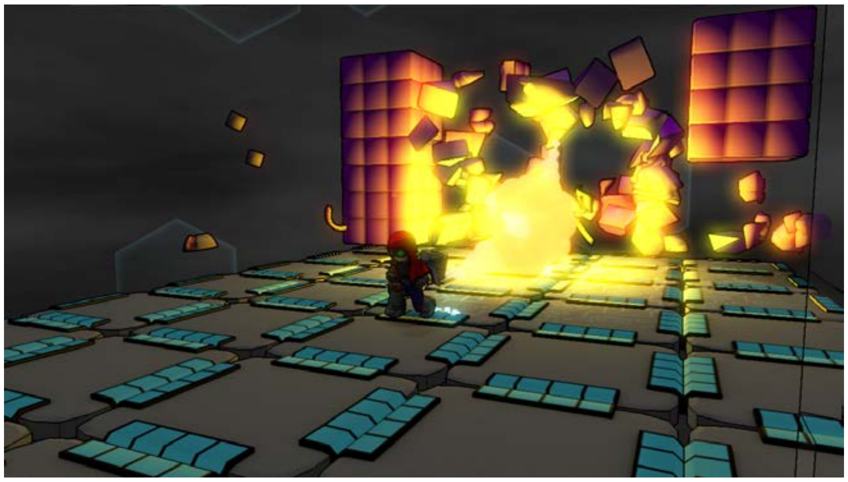


Figure 91: Gradient additive lighting used for an explosion effect

Like games in the *Tales* series, *Xeero* gives objects a black border (though, *Xeero* indiscriminately gives *all* objects a border, rather than just characters and props, since the props *are* the environment).

4.6. Animation

In *Xeero*, any change in the visual state of an object is done through an animation; very few objects are allowed to change without doing so smoothly over a period of time. All animations involving movement take realistic physics into account, whether these objects are directly simulating physics with acceleration, velocity, and forces or are simply simulating the resultant movement created by these physical phenomena. *Xeero* makes liberal uses of damped springs in the game to allow objects to smoothly snap (and potentially bounce) into place, whether it is camera motion, opening and closing gates, or floating, telekinetic weapons.

With few exceptions, skeletal animation in *Xeero* is restricted solely to the player character and the malware the player combats. For these animations, the primary focus was to give realistic properties to unrealistic motion. The player may be able to float in mid-air, telekinetically grab a block object, or spin in mid-air, but the properties of realistic motion, like Disney's 12 Principles of Animation (Thomas and Johnston), are considered for each motion. Most animations attempt to telegraph their motion with necessary windup, give a sense of weight to the action with appropriate follow-through, and attempt to make the objects feel like actual, physics constructs with overlapping actions and considering the inertia of the object's component pieces. This helps establish satisfying motion for character actions, typified in the games of the *Kingdom Hearts* series.

Additionally, each character has a large number of motions, and the player is able to perform actions in rapid succession (see **Mechanics - Attacking and Defending**). Thus, the characters need to be able to quickly and seamlessly transition from one animation into another. Partially, this is accomplished with a small number of transition animations, specifically leading one animation into another, but the majority of animations work from a given idle pose of the character into whichever action the character will perform. To avoid any unattractive "snapping" from one pose to another, an animation blending system was developed (see **Animation**

Blending), to allow any arbitrary number of animations to blend and transition into any other arbitrary number of animations.

4.7. Process - 3D

The following section will describe the process and software Eric used to create some of the 3D assets used in *Xeero*.

4.7.1. Software Used

For the creation of 3D assets, Eric used Adobe Photoshop, Blender, and ZBrush. Adobe Photoshop, photo and image editing software, was used for building modeling sheets during conceptualization and for building textures for models. Blender is a full 3D pipeline application, with features from the initial stages of modeling and sculpting to skinning and animation. The 3D application was used in modeling, texturing, rigging, skinning, and animating models. ZBrush, a digital sculpting tool, was used to sculpt some characters.

4.7.2. Characters and Props

Most characters and props created for the game follow a similar process: conceptualization and sketching, modeling, sculpting, texturing, rigging and skinning, and animation, finalizing each step before moving onto the next.

Concept and Sketches

First the concept for the object or character is created. For characters, this usually involves defining unique characteristics and features for the character, and firming the design through several rough sketches. In this step, most of the details for the character, like clothes and ornaments, are represented on paper.

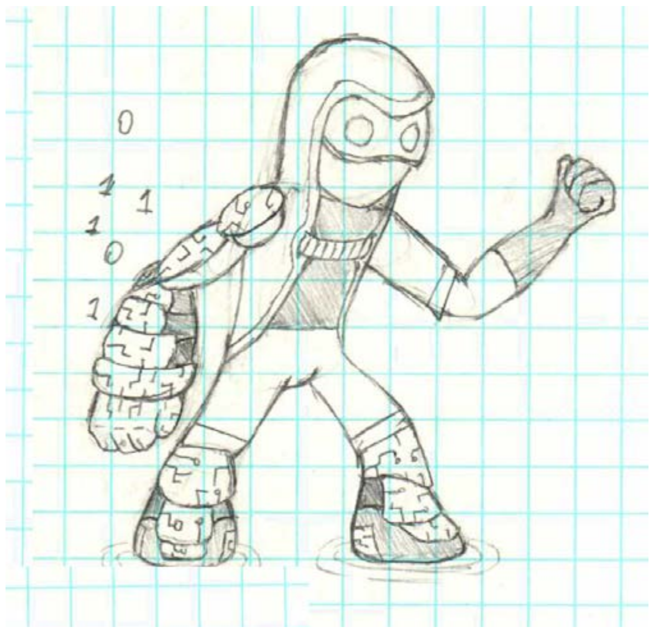


Figure 92: An early sketch of Xeero

Next, the proportions for the character are precisely defined. Eric creates a modeling sheet that will be used to model the character. Usually, in this step, paper sketches are scanned and a modeling sheet is created in Adobe Photoshop.

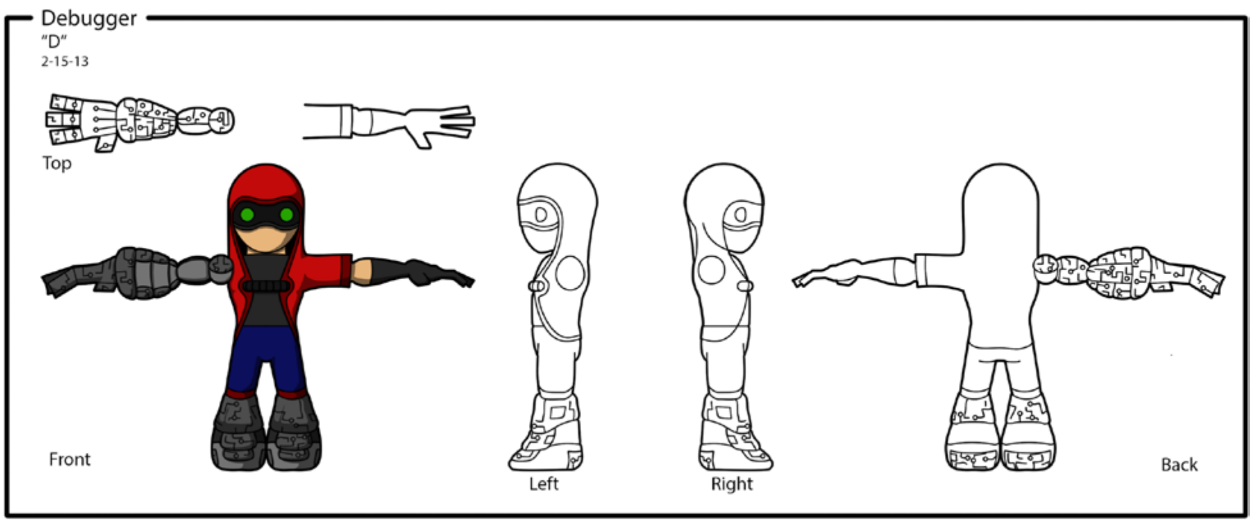


Figure 93: The modeling sheet used for Xeero

Modeling sheets usually contain several views of the character -- front, profile, or back view -- and any additional equipment the character may use, like a weapon.

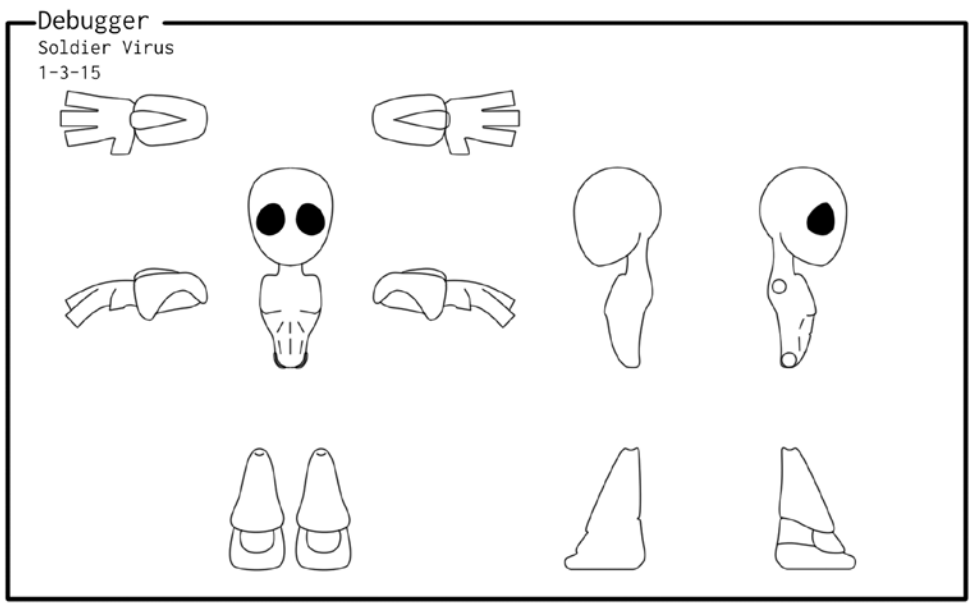


Figure 94: The modeling sheet used for the soldier virus

For most props, the necessary functionality has already been defined in the terms of gameplay and mechanics, and this step involved giving form to the function. Most props in *Xeero* are based on real computer hardware, redesigned for *Xeero*. For those props, sketches or modeling sheets aren't usually created, and reference images are gathered instead.

Modeling

After modeling sheets have been created for characters, these are imported into Blender, mapping each view in the modeling sheet to the corresponding view in Blender, so the proportions of the model can be easily compared to the modeling sheet and adjusted. Here the general proportions for the model are created through box and extrusion modeling.

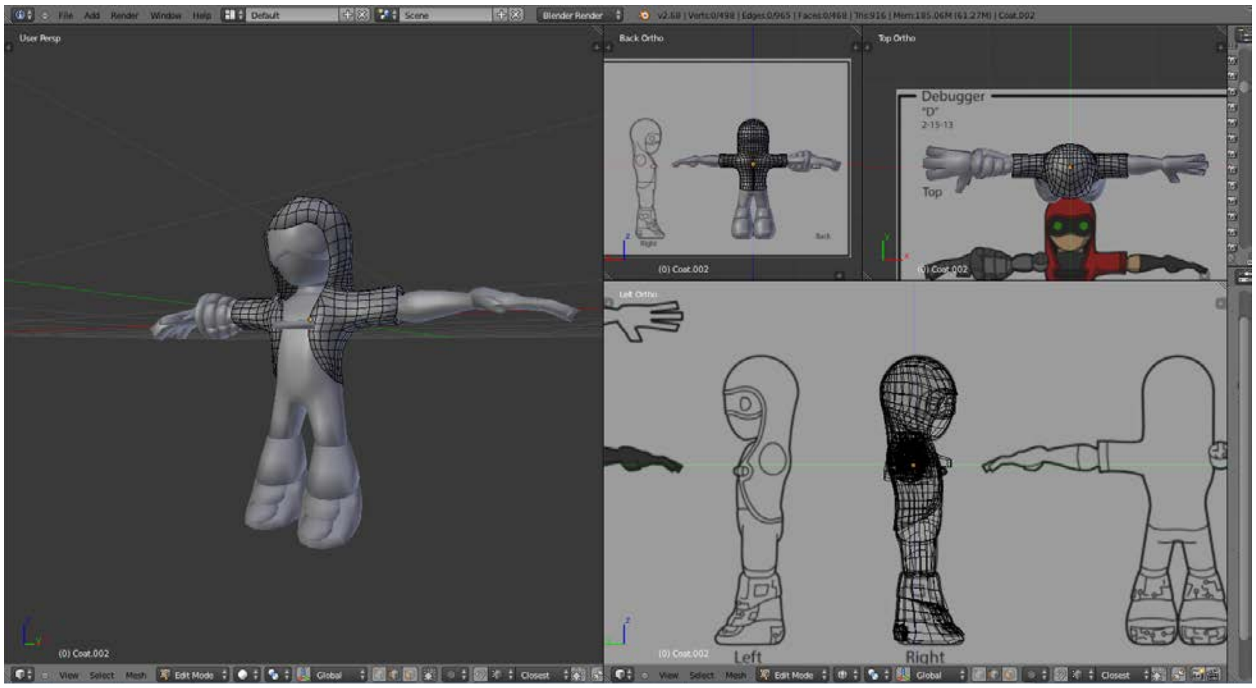


Figure 95: Modeling Xeero from the modeling sheet

Most props use mostly hard surfaces using more precise angles and lengths, so their geometry is entirely modeled in Blender. For props, there's usually more experimentation with forms, since the final design object is mostly undefined.

Sculpting

Some characters were modeled in Blender first, then imported into ZBrush, while others were created entirely in ZBrush. When models are created exclusively in ZBrush, a similar method is employed: the modeling sheet is imported and mapped to the appropriate views and the general forms of the model are created.

After the general forms have been created, either through ZBrush or imported from Blender, shapes are refined and details added, while ensuring that most shapes stay simple and minimalistic, as mentioned in the **Style - 3D** section. After the model has been shaped and refined, they are retopologized to meet the vertex count requirements for the game, and are exported to Blender.

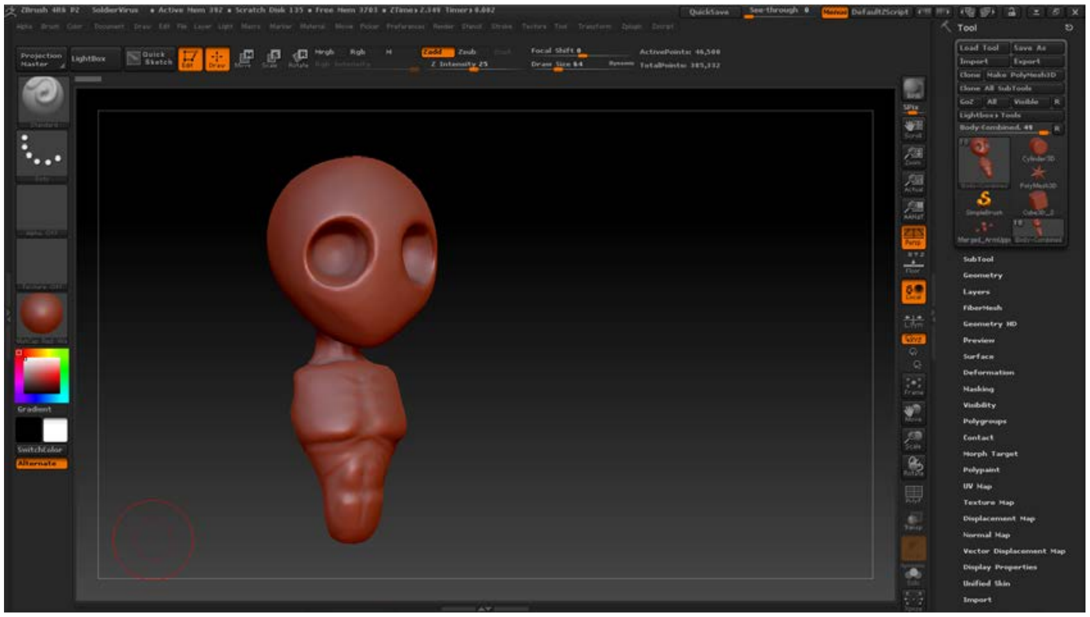


Figure 96: Sculpting the torso for the soldier virus

Texturing

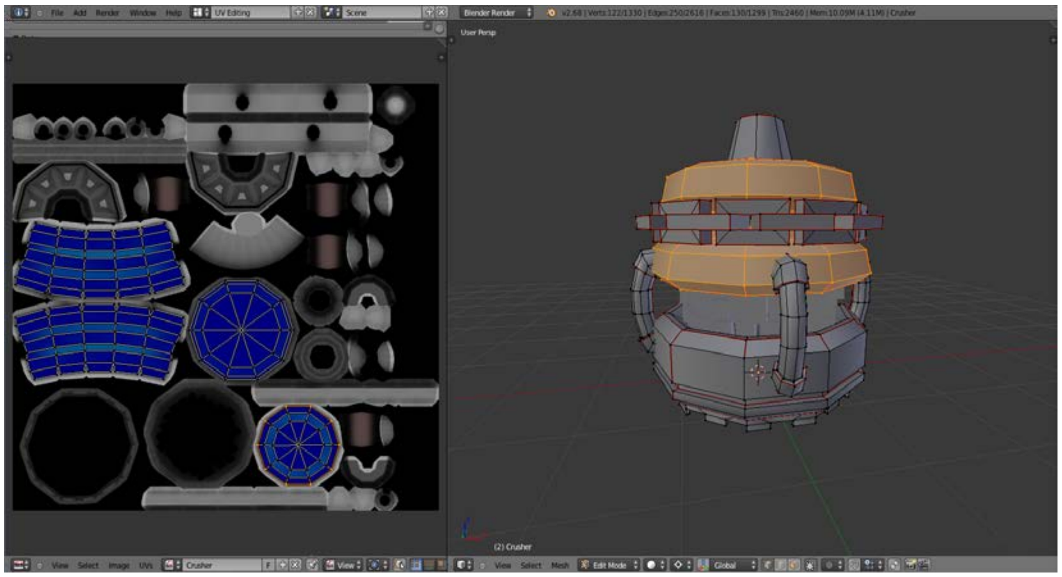


Figure 97: The UV seams for a prop

In Blender, seams and UV maps are created and assigned to the imported character or modeled prop. For some props, extra consideration is given for certain effects; some props change the color of some of their materials as part of an animation in-game. Sections of the model that will be used for the same effects are grouped into single textures.

UV maps are exported and loaded into Photoshop where the colors and shading are defined. For most surfaces, flat colors with soft shading is applied directly to the texture (see **Style - 3D**).

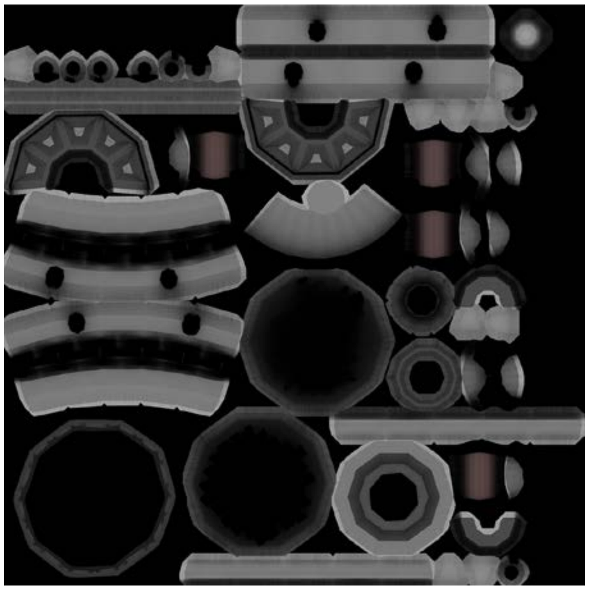


Figure 98: A texture map for the compressor, soft shading is baked into the texture

Some models use “tech lines,” or glowing lines that appear when certain actions occur in game (pictured right). Other objects also “erode” as they become more corrupted, becoming fragmented (pictured left). Both types of information, where the tech lines appear and what parts of the model get eroded, is defined and encoded within the textures generated for a model.

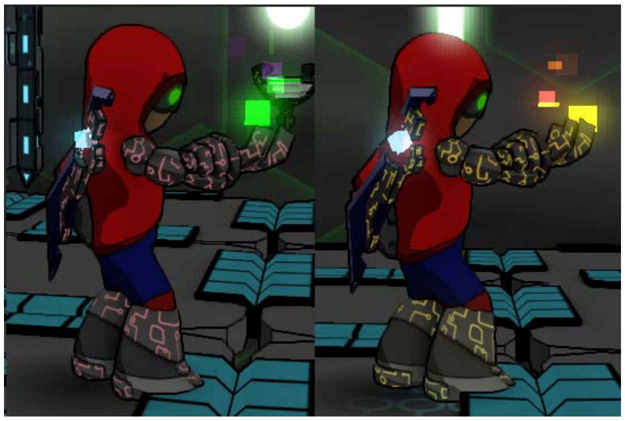


Figure 99: Xeero using different "tech line" colors

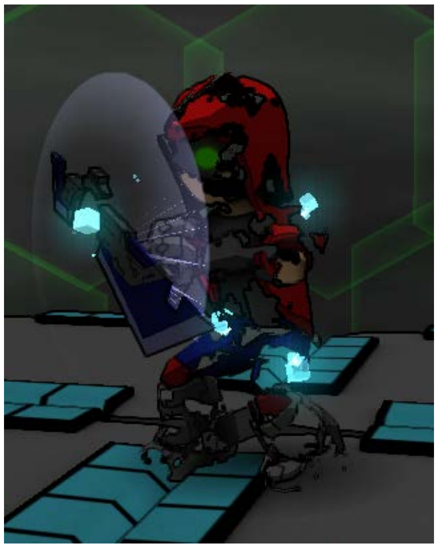


Figure 100: Xeero, eroded from low health

For most props in the game, this completes the model, and it is exported and used in-game. For characters and some props, they need to be rigged and skinned so they may be animated.

Rigging and Skinning

At this stage, the model has been finalized: the geometry and textures have been defined and the models are anticipated to require no or minimal changes.

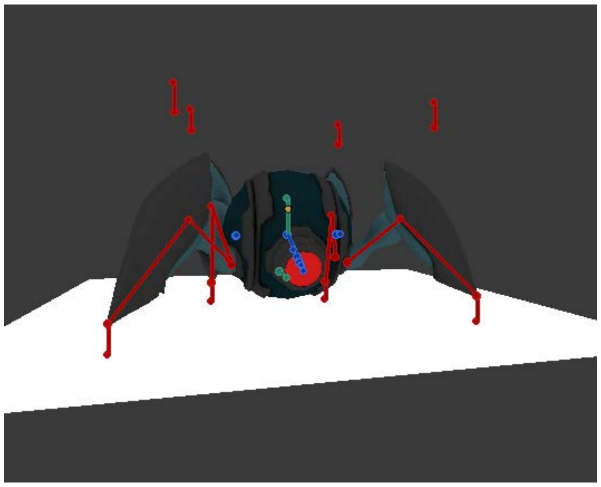


Figure 101: The skeleton used for the spider virus

Eric uses Blender's rigging tools to define a skeleton for the character or prop. Most models use either deform bones (bones that directly move vertices on the model) or inverse kinematic positioning bones (bones that define the end position and joint direction for deform bones that use inverse kinematics).

Next, the character mesh is skinned, and the model's vertices are heat-mapped to the bones of the skeleton. From there, Eric weight paints (determines how much influence any bone has on a single vertex) the models to correct errors in the heat-mapping and to ensure that joints and folds in the mesh are correctly deforming.

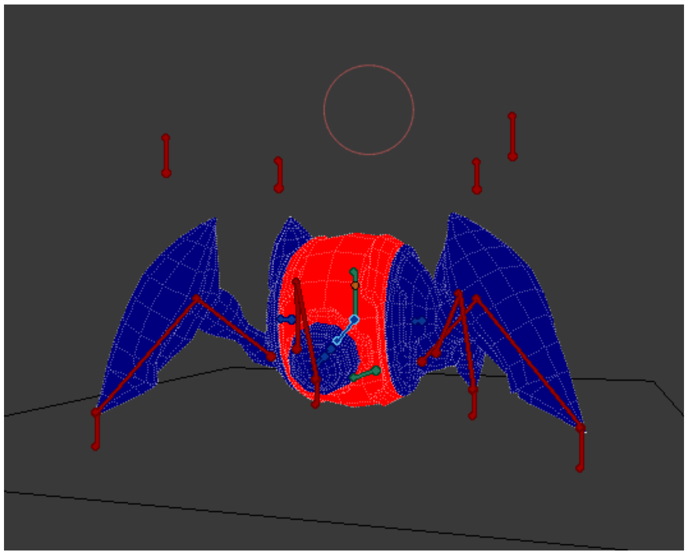


Figure 102: Weight painting bones for the spider virus

After testing, tweaking, and weight painting, the model is ready for animation.

Animation

Usually, after a model has been skinned, Eric first creates poses that serve as the basis for most of the animations. Usually, at least an “idle pose” is created and stored for the character. When creating a new animation, usually the idle pose is used for the first frame, and the character transitions into the major poses from them.

Next, Eric uses Blender’s tools to create animations for the model. In Blender, specific animations are broken into “actions,” each action containing the key frames for the animation.

Eric creates animations for a model first by defining the key frames for the major poses of the action and timing out each pose over approximately how much time the animation should take. Some animations are precisely defined in their timing (like jumping and landing animations for enemies, which follow a template) while others allow for more flexibility. After major motions are defined, windup and recoil frames are added, and frames are adjusted to account for overlapping action.

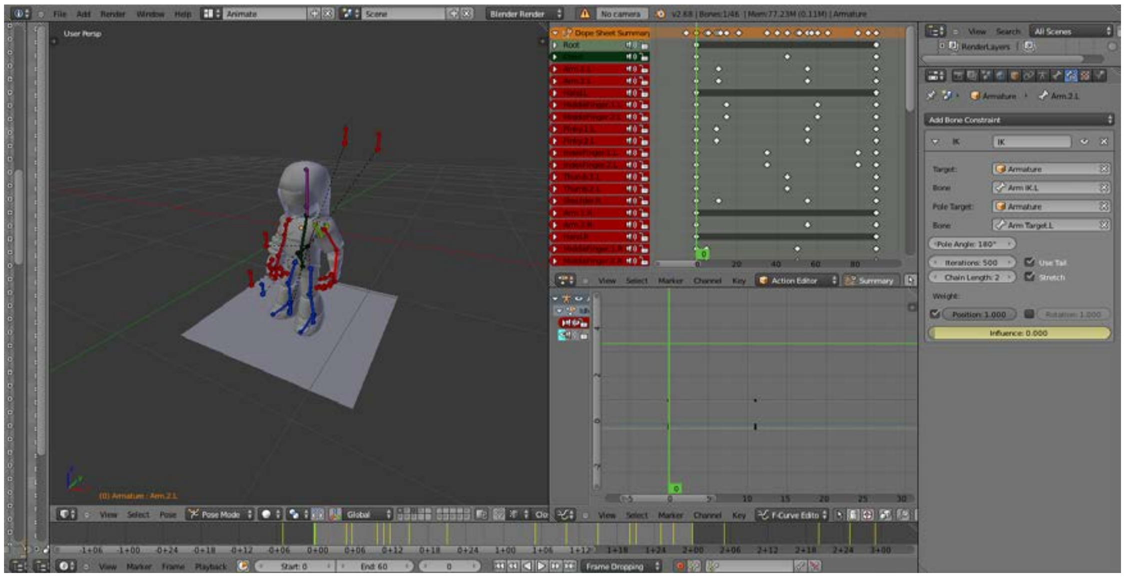


Figure 103: The pose and key frames used in the "idle" animation for Xeero

Animations are usually tested in-game one at a time as they are completed. The model and its animations are exported, imported into *Xeero's* engine, and tested.

4.7.3. Debris Model

Some objects in *Xeero* are destructible, and after they are destroyed, they crumble into rubble. Each piece of rubble is created as a separate model, and each piece is loaded into the game to replace the destroyed object.

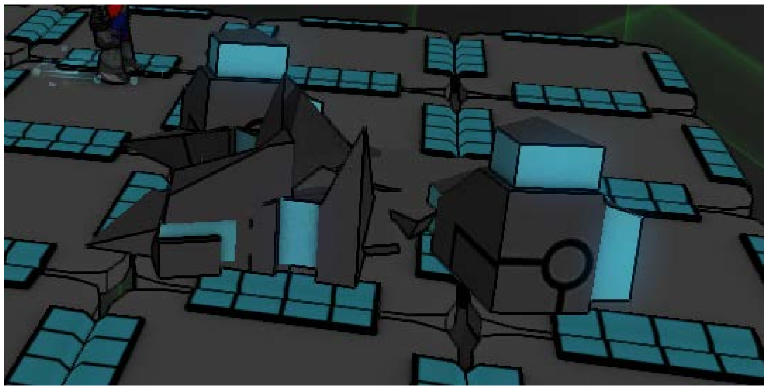


Figure 104: A pile of rubble a block left behind

Each debris model is created from another finished model. The model is divided along where the breaks occur, and new faces and texturing seams are added to fill the volume of the object. Each debris piece is UV unwrapped, and textured to match the original texture of the object. Then each piece is exported and loaded into the engine.

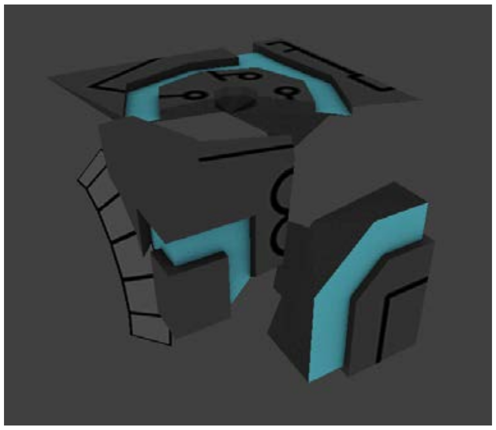


Figure 105: The individual pieces of block debris

4.8. Style - 2D

Two-dimensional art plays a supplementary role in *Xeero*, reserved mostly for icons, GUI elements, and particles. Nevertheless, 2D art follows the themes of "stylized art" used for 3D models.

Building the aesthetic for 2D art in the game was simple after the 3D style was cemented, helped by the fact that the 3D art style was originally conceived to mimic 2D art. Most 2D elements used vibrant, solid colors with thick black borders.

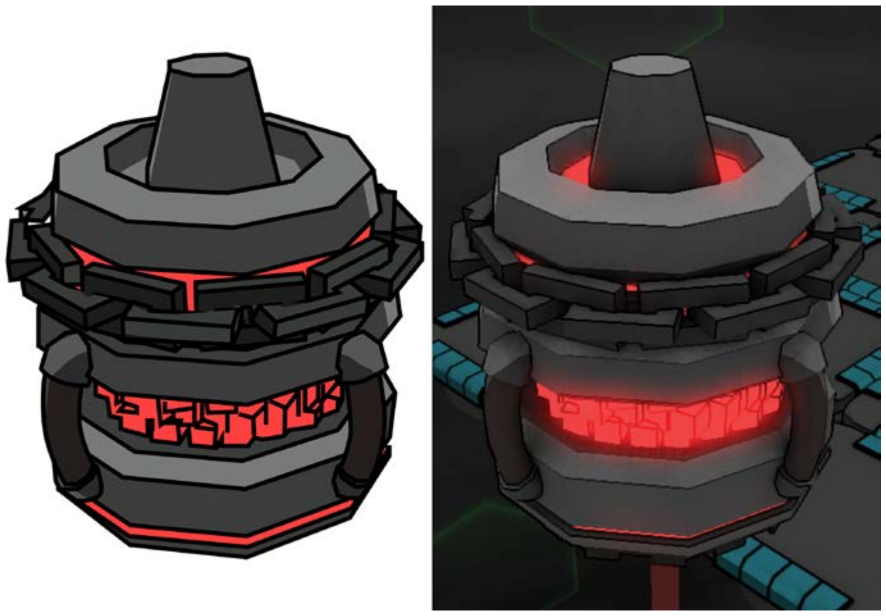


Figure 106: A 2D icon and its 3D counterpart

Two-dimensional art was created in both Adobe Illustrator and Photoshop in equal measure. Illustrator's vector art was used to create most of the icons for in-game objects, and Photoshop was used for bitmap effects and particles

Most fonts used in the game were sourced externally, using the open licenses of some fonts (like Raph Levein's Inconsolata and Typodermic's Quadrangle). Some fonts in the game used effects that required additional processing on the original OpenType or TrueType font files. In Microsoft XNA, the framework used to create the *Xeero's* engine (see section **Technical Development - Frameworks and Platforms**), fonts can be turned into game assets directly from font files or by loading in a texture containing all of the characters used in the font. Leveraging this feature, Eric used a tool called SpriteFont 2, created by Nubik, allowing him to recreate any (open license) TrueType or OpenType font as an image. Then, Eric could use Photoshop to process the fonts and add additional effects, like borders.

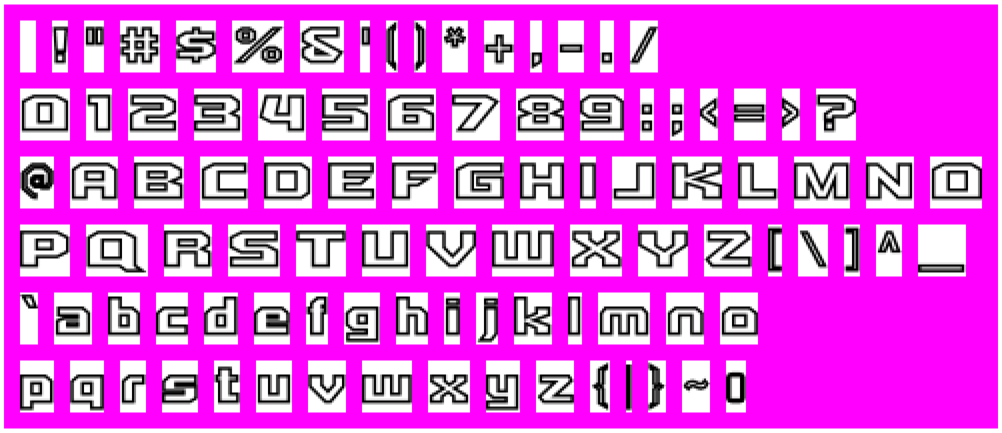


Figure 107: A font texture generated by Spritefont 2 and modified in Photoshop

Additional consideration was also needed for certain types of particle effects. Some effects, like the firewall that extends below the world and **Purge Gates** that can stretch to cover an arbitrary area (both pictured below), required special textures that could be both be tiled and dynamically processed.

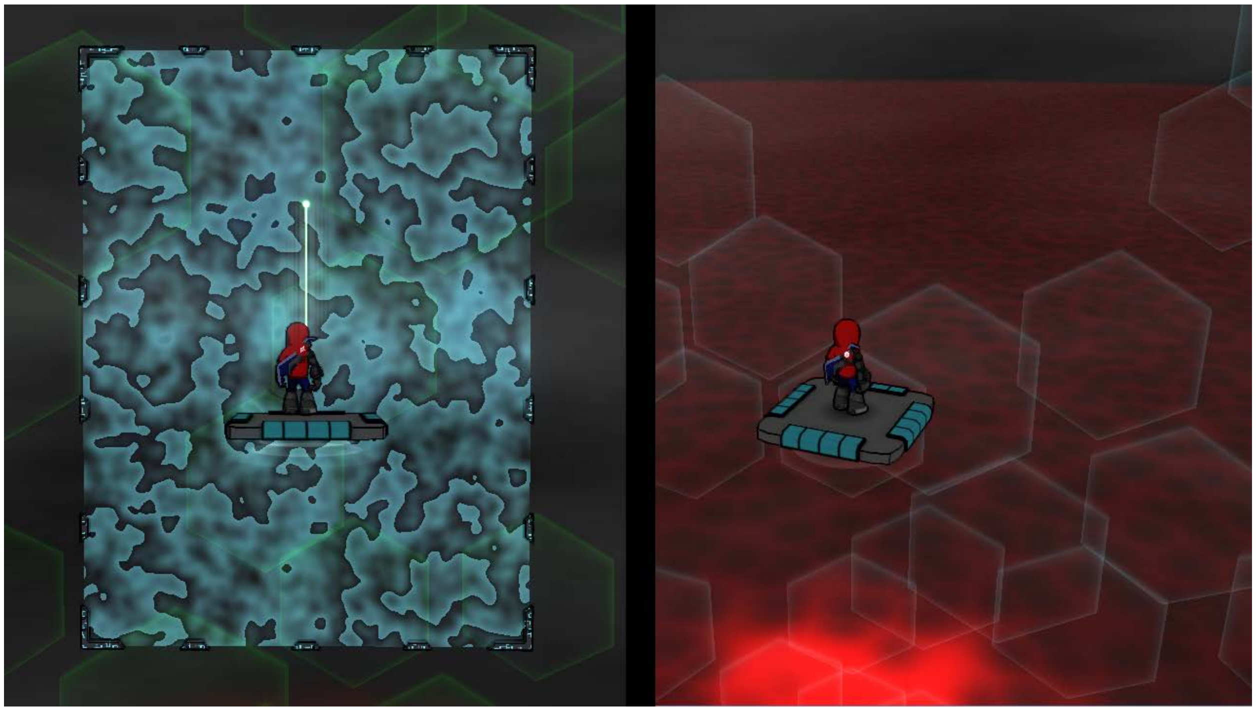
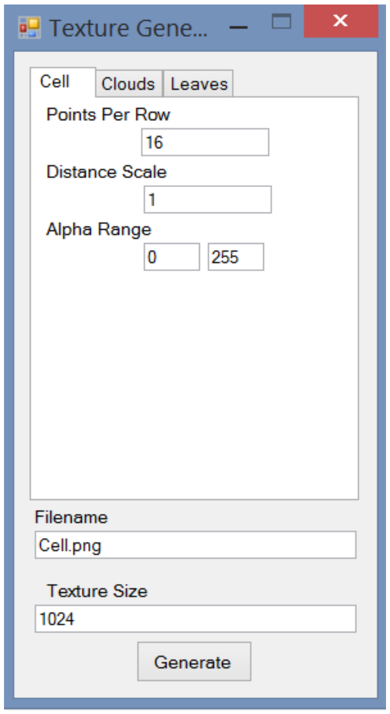


Figure 108: Effects in Xeero requiring special, tile-able textures

For both of these textures, Eric created a program to generate procedural textures using different equations.


Texture Gene...

Cell | Clouds | Leaves

Points Per Row: 16

Distance Scale: 1

Alpha Range: 0 255

Filename: Cell.png

Texture Size: 1024

Generate

Figure 109: The interface of the procedural texture generator

This program creates tile-able textures using either the "Cell," "Clouds," or "Leaves" equations. Each equation can be tweaked, changing the resolution, density, or other properties of the generated image. Different textures generated by the program are below.

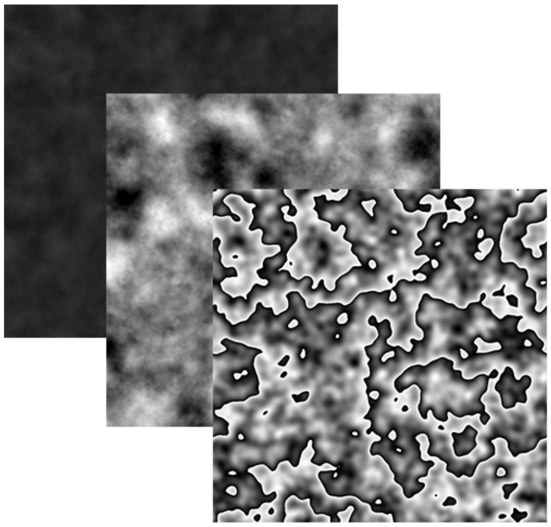


Figure 110: Some sample textures generated by the procedural texture generator

4.9. Environment

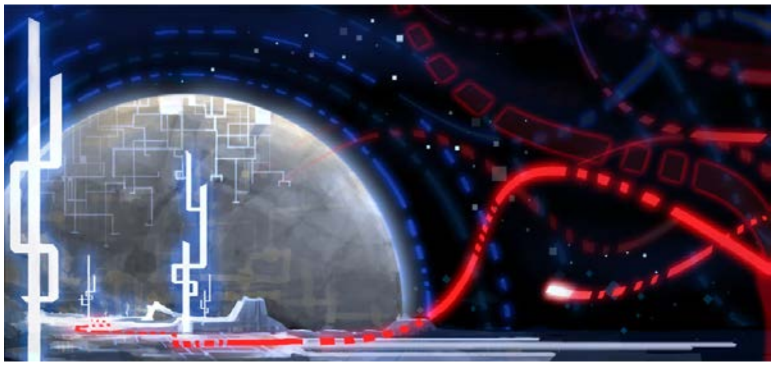


Figure 111: "Cyberspace" by TypoCity

Inspiration for the general tone of the world of *Xeero* was drawn from several independent artists, among them, Joou Chiyanmon, Justice Von Brandt, and TypoCity from DeviantArt. These artists had created "cyberspace-"themed art that inspired the use of blacks with splashes of vibrant color for backgrounds in *Xeero*. Cyberspace is often depicted as a "void," with little populating the space except objects of interest.

These inspirations are clear in the levels and skyboxes used in *Xeero*. As mentioned in the **Level Editor** section, levels are built modularly out of specific, gameplay-altering modules

Figure 112: "Cyberspace" by Justice Von Brandt

allowing for minimal set pieces. The player, standing on one platform surrounded by only a few others, appears to be standing in a void.

In all of the levels of the first hub, backgrounds are black and gray clouds. Lining the background like pieces of a massive cylinder are dozens of large, glowing, slowly rotating hexagons; the colors of the hexagon change with the level, picking randomly from blue, green, or yellow. As the player approaches bugs in each level, the hexagons glow a deep red and spin faster. Below depicts the skybox used in *Xeero* and one of the images that inspired it.

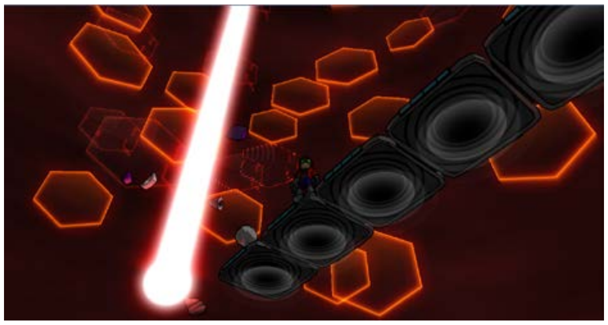


Figure 113: The skybox in Xeero

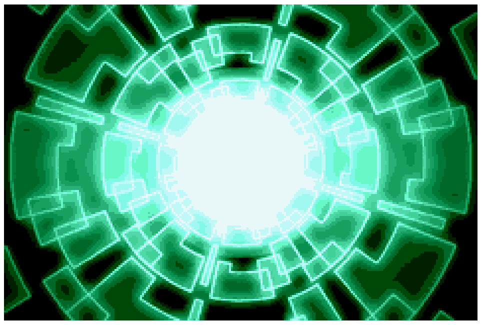


Figure 114: "Cyberspace Move" by Joou Chiyanmon

The theme of the first hub is “debris,” and the player is allowed to view the damage the Alpha Virus does to the system. The hub is made of several shattered islands and an abundance of swirling debris. This structure is inspired by the depiction of ruins by independent artists like Ninjatic and Joel Faber of DeviantArt. In the image shown below, these desolate worlds are composed of individual, broken islands.

Figure 115: Floating Ruins by Ninjatic

4.10. Objects

The following section describes the specific motivations for some of the gameplay objects (see **Design - Objects**) used in the game.

The objects in the game are designed to give form to the abstract elements “inside of a computer.” For some objects without a direct analog to a piece of hardware or software (like **Platforms, Blocks,** and **Leap Targets**), determining how to represent them was a challenge.

For Blocks and other reusable objects, Eric chose a specific type of “material.” The objects have an abstract, “technological” appearance, using hard edges, dark colors, and angular, irregular shapes. Each digitizable object has an associated color (e.g. blocks are blue, bombs are red): the color appears on the model and all particle effects using that object are tinted to match. Additionally, all digitizable objects have prominent “tech lines” (see **Style 3D - Texturing**).

For other objects, a theme quickly emerged, one using actual computer hardware to inspire the creation of game objects.

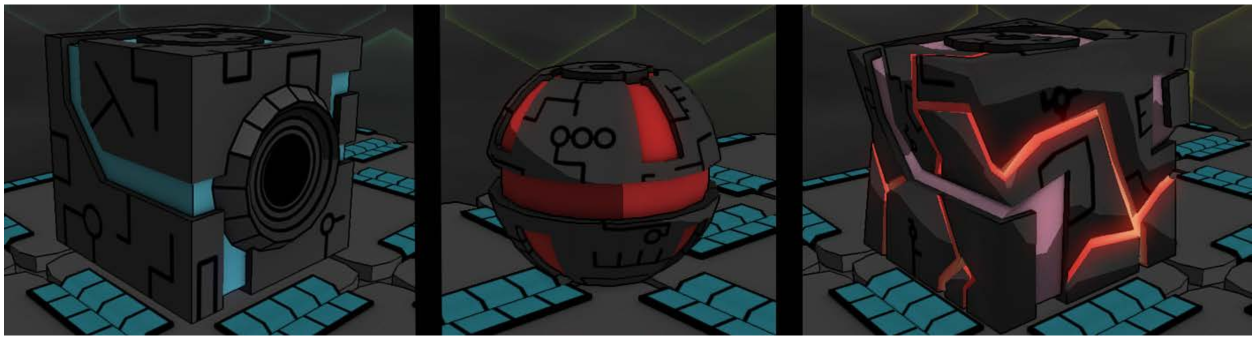


Figure 116: Different digitizable objects and their colors (block are blue, bombs are red, unstable blocks are a mix between the two)

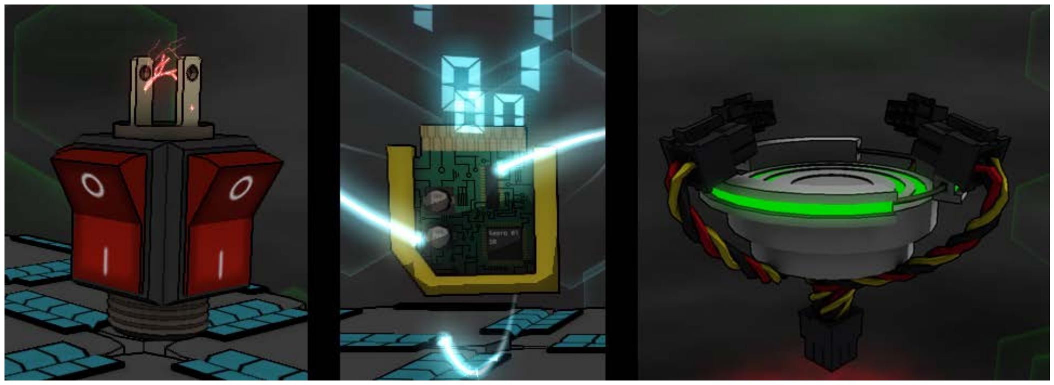


Figure 117: Some hardware-inspired objects in Xeero

Among others, on-off switches were used to inspire **Hit Switches,** power buttons were the basis for **Buttons,** and PCI-E connectors and wires were used to create portals to other programs.

Other objects were based on icons and symbols used in software. **Breakpoints**, which act like checkpoints for players, are based on breakpoints--which stop the execution of a program wherever the breakpoint is located--used in several integrated development environments used by computer programmers. **Archives** and **Big Archives**, which act like treasure chests the player can open, are based on icons traditionally used to represent databases--structures containing organized data. Representing objects using actual hardware or iconography was a strategy designed to make it easier for players to determine what game objects would do.

With other objects, the intention was to use the computer motif to create a "digital redesign" of familiar items. The portal used to transport the player to the boss fight (pictured right) is based on other large gateways or portals seen in fantasy settings, but it has designed with PCI-E connectors, pipes, and the types of hard edges and angular shapes used with digitizable objects. Similarly, **Bit Slicers**, a platforming obstacle for the player, are based on familiar bladed traps.

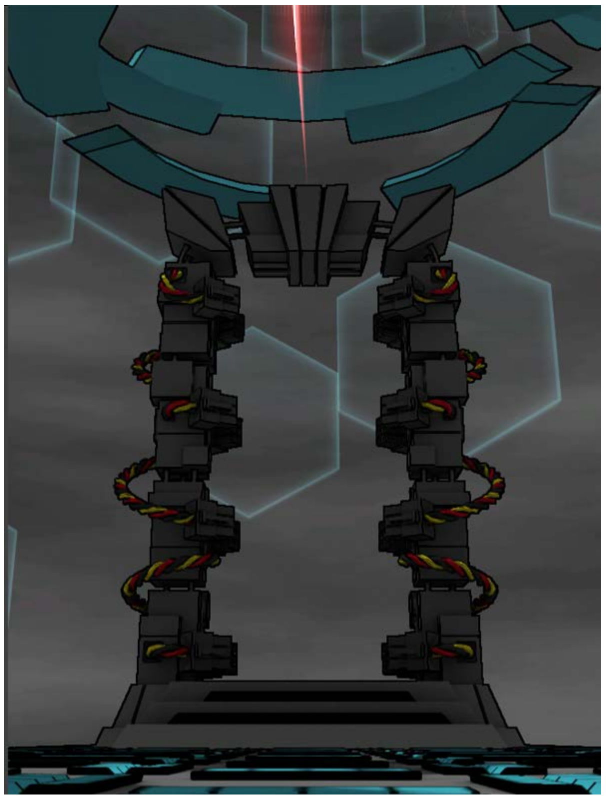


Figure 118: The boss door, a digital redesign of a large portal

4.11. Promotional Artwork

While Eric created the visual art used *within* the game, the team collaborated and sourced help from another student to create promotional artwork for *Xeero*. Within the team, Anthony created a series of trailers for the game (see **Production Management - Trailer**). Externally, Kedong Ma, an IMGD art student, was contacted to create promotional artwork for *Xeero.* Ma collaborated with the team, sharing game assets and iterations on artwork. Over the course of seven weeks, Ma created an animated 3D logo and poster for the game (see **Appendix B**).

5. Audio

5.1. Sound Effects

In the following section, we will discuss the design process and implementation of sound effects in *Xeero*. We will go into details on the different programs that Dan utilized, how he found source materials, and his editing process. We will then discuss the steps he took in order to implement the sounds into the game.

The first thing Dan needed to start designing the audio was a list of the different sound effects that would be needed for the game. During their first semester on the project, Eric walked Dan through the game, identifying everything that was needed audio cues, including animations, user interface cues, ambient sounds, and particle effects.

Dan added to the list all of the sound names (see **Appendix C**). This sound asset list would act as a vital reference for the remainder of the project, helping Dan and Eric monitor each sound cue name, whether the sound should loop, if the file had been created, and if the file had been inserted into the game engine.

For most of the sound cues, Eric gave Dan a general idea of what type of sound he had in mind for different sources. For other sounds it was up to Dan to come up with an original idea. Dan kept a separate list of requests and suggestions from Eric for when he would design the sounds. This reference guide would help Dan to focus in on a single sound effect.

Dan developed a color-coding system for the list to rank the importance and priority of how quickly the sounds needed to be made. Green cells had high importance, as they were for sounds that still needed to be created and were very common in gameplay, such as character movements and primary mechanics. Dark green cells had not been made yet, but were of low importance. Yellow cells were for sounds that had been made already, but still needed some minor adjustments. Red cells were for sounds that had not been made, and still needed to be discussed between Dan and Eric. Clear cells were for sounds that were finished, polished and had been given Eric's seal of approval. Finally, blue cells were for sounds that Dan had made, but still needed Eric's approval.

Now Dan was ready to begin the actual designing. Since Dan had very little experience with recording, he did not have a very large sound library to utilize. This gave him few source materials for the different sound effects. Since the game's setting is a digital domain, most of the sounds would be heavily computer oriented. Dan posited that it wouldn't make sense to do much recording, as the sounds would come off as too organic. This still did not solve the problem of how he would obtain source materials.

Dan decided to look for most of his source materials from free sound libraries. Under Creative Commons, these sources files were royalty free, and Dan was free to edit and customize the different sounds as he saw fit. Using the few sounds Dan had in his own library, and the different sounds he found in the online libraries, he was ready to actually begin the editing process.

The sound editing program that Dan had the most experience with was Pro Tools, but having access to this program was out of his budget. During this time, Dan was acting as teaching assistant to Keith Zizza, a Professor of Practice for game audio in the IMGD department. Zizza was having his students utilize Reaper, a digital audio workstation (DAW) released by Cockos.

The advantages of using Reaper in the classroom setting was that it was free to download, and had a simple user interface. Students could download the DAW on their home machine and could continue to easily work on assignments both on and off campus. Zizza recommended to

Dan that he should take the time to learn Reaper, as it not only worked very similarly to Pro Tools, but also fit well into Dan's budget.

Dan downloaded the software onto his home machine and began to familiarize himself with the interface, as well as take in some tips from the online user manual (Francis, 2014). Just as Zizza had stated, there were many similarities between Reaper and Pro Tools. This allowed for many of Dan's editing skills to be transferred over to this new program.

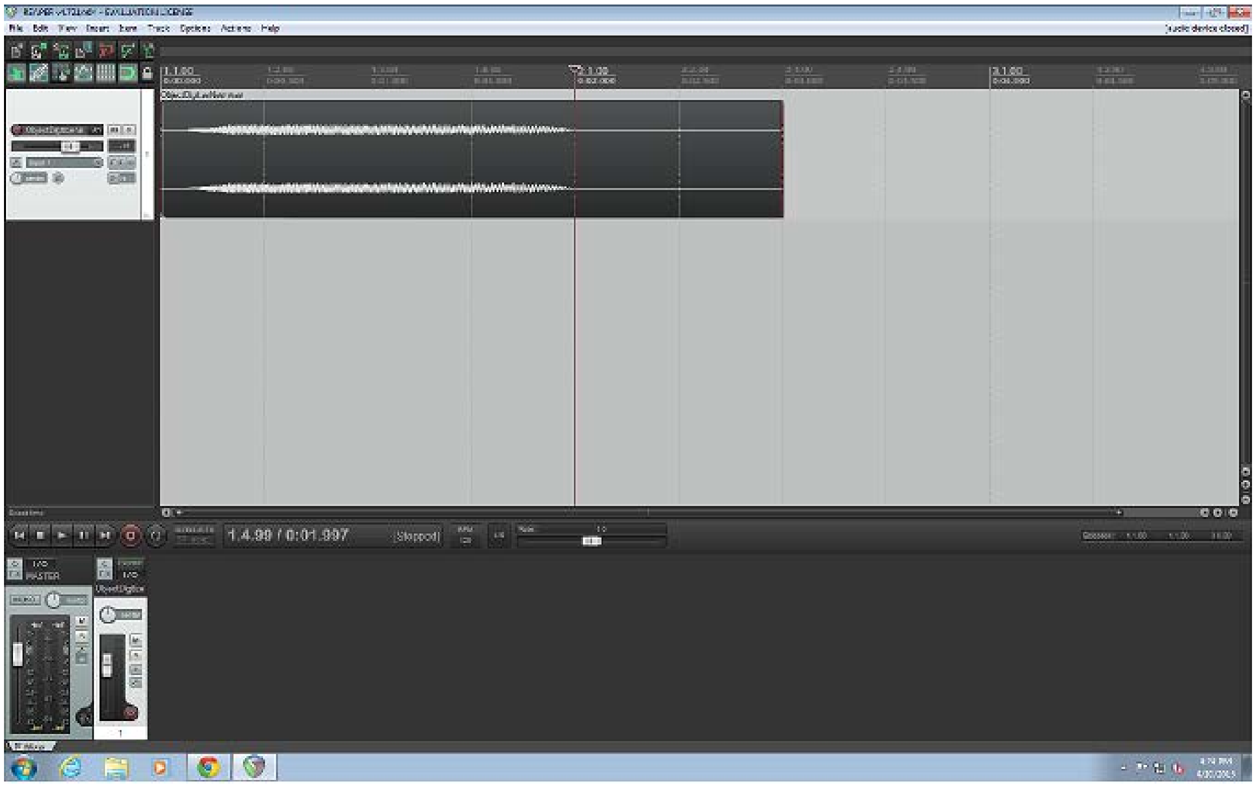


Figure 119: Screenshot of the main Reaper interface

The initial goal for Dan was to work through the sound asset list as quickly as possible. Each sound cue would be filled with a temporary sound, insuring that the game was always in a presentable state. Later in the project, Dan would go through and replace these sounds with more polished and finalized versions. Dan's intent was to make each sound as close to perfect as possible, with hopes that the sound might not have to be changed later in the project.

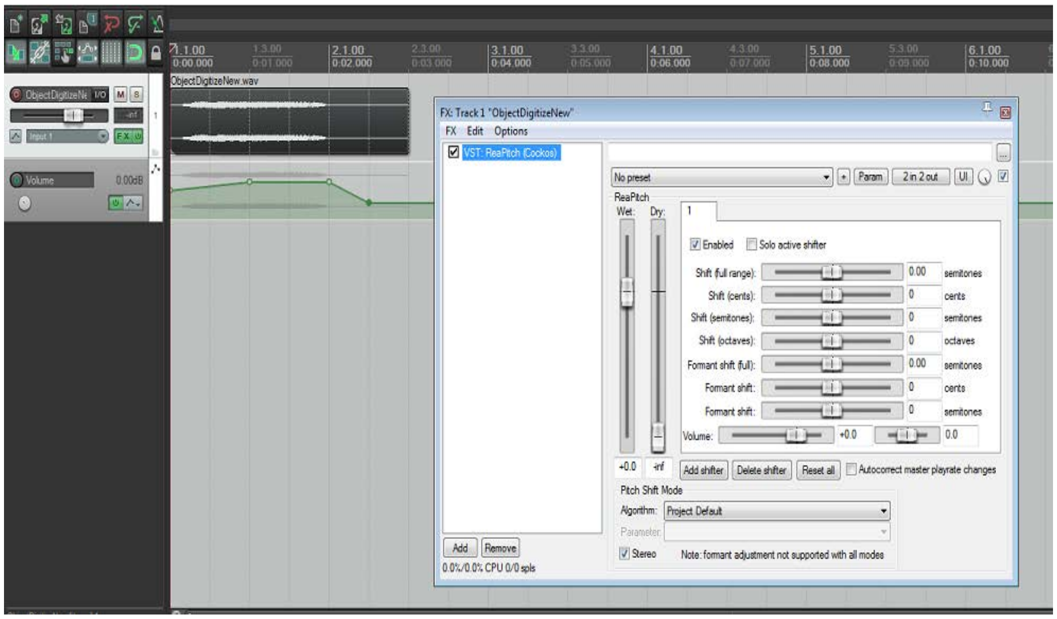
Much of the design process was experimental. Since most of the sounds *Xeero* needed were fictional, there weren't many real-world objects to use for reference. The goal of each of Dan's work sessions was to make something that sounded as artificial as possible, but that meant working without a specific sound in mind.

Often, Dan would insert a source file into Reaper and then mess around with the different available plug-ins. During these sessions, Dan often wouldn't have a set sound cue in mind. He would just mess around in the workstation until something struck him as interesting, and then would look through the asset list to see if it would fit anywhere in the game. This method encouraged Dan to experiment more with the Reaper interface, though it was not the most efficient method of production.

Occasionally, sounds Dan would design would be decent for sound cues that weren't the highest priority. While all of the sounds would need to be designed at some point, some of the more important cues were being pushed further back into production.

When experimentation wasn't enough, Dan could go by the list of suggestions that Eric had provided for him. Since the list had some generic directions on them, Dan could just find a suitable source file, and tweak it to Eric's specifications.

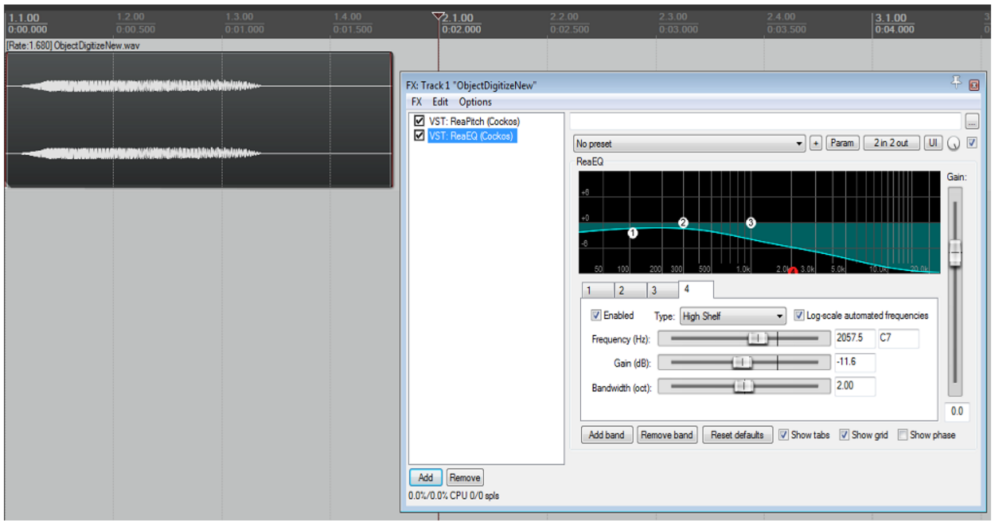
In terms of plug-ins, the pitch modifiers often provided the most interesting results. By shifting the octaves up and down, it would often distort the source sounds well enough so they would fit with the theme of the game. Reaper also has a tool that raises the playback speed of the sound, but does not alter the pitch. This compression allowed for Dan to easily shorten sound cues that could fit in time to the different animations in the game.



Enabled
Shift (full range): 0.00 semitones
Shift (cents): 0 cents
Shift (semitones): 0 semitones
Shift (octaves): 0 octaves
Formant shift (full): 0.00 semitones
Formant shift: 0 cents
Formant shift: 0 semitones
Volume: +0.0 0.0
Pitch Shift Mode
Algorithm: Project Default
Stereo
Note: formant adjustment not supported with all modes

Figure 120: Screenshot of Reaper, using the Pitch modifier plug-in and volume modulation

Any sounds that had extraneous noise or frequencies could be altered using equalization. Using this process, Dan could make it so only specific frequencies of the sound could be heard. Since some of the source files were not of very high quality, it helped to eliminate extraneous sounds.

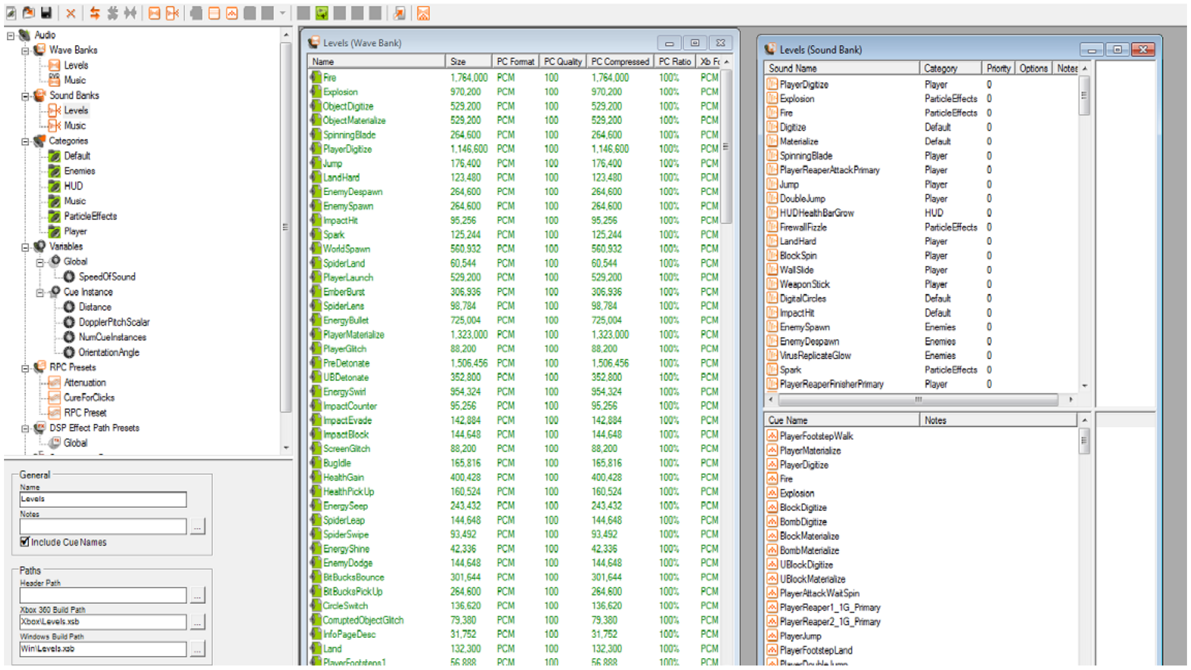

FX: Track 1 "ObjectDigitizeNew"
FX Edit Options
VST: ReaPitch (Cockos)
VST: ReaEQ (Cockos)
No preset
Param
2 in 2 out
UI
ReaEQ
Gain:
Add band Remove band Reset defaults Show tabs Show grid Show phase
Add Remove
0.0%/0.0% CPU 0/0 spls
[Rate:1.680] ObjectDigitizeNew.wav

Band 4
Enabled
Type: High Shelf
Log-scale automated frequencies
Frequency (Hz): 2057.5 C7
Gain (dB): -11.6
Bandwidth (oct): 2.00

Figure 121: Screenshot of Reaper, where the sound speed has been compressed and higher frequencies have been attenuated.

After the sounds were edited, they were ready to be exported. The sounds would be rendered to 16-bit wav files, with a 44,100k sampling rate. After listening to the finished files a few times, Dan could now import the sounds into the audio engine.

Since Eric was building *Xeero* using the Microsoft XNA engine, the XACT audio engine was necessary for integrating the sounds into the game. Dan had never worked with XACT before, so he needed to do some research to become familiar with the interface. This would also involve reading through the formal XACT user documentation (Microsoft, 2015)



General
Name: Levels
Notes:
[✓] Include Cue Names

Paths
Header Path:
Xbox 360 Build Path: Xbox\Levels.xsb
Windows Build Path: Win\Levels.xsb

Levels (Wave Bank)

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb F
Fire	1,764,000	PCM	100	1,764,000	100%	PCM
Explosion	970,200	PCM	100	970,200	100%	PCM
ObjectDigitize	529,200	PCM	100	529,200	100%	PCM
ObjectMaterialize	529,200	PCM	100	529,200	100%	PCM
SpinningBlade	264,600	PCM	100	264,600	100%	PCM
PlayerDigitize	1,146,600	PCM	100	1,146,600	100%	PCM
Jump	176,400	PCM	100	176,400	100%	PCM
LandHard	123,480	PCM	100	123,480	100%	PCM
EnemyDespawn	264,600	PCM	100	264,600	100%	PCM
EnemySpawn	264,600	PCM	100	264,600	100%	PCM
ImpactHit	95,256	PCM	100	95,256	100%	PCM
Spark	125,244	PCM	100	125,244	100%	PCM
WorldSpawn	560,932	PCM	100	560,932	100%	PCM
SpiderLand	60,544	PCM	100	60,544	100%	PCM
PlayerLaunch	529,200	PCM	100	529,200	100%	PCM
EmberBurst	306,936	PCM	100	306,936	100%	PCM
SpiderLens	98,784	PCM	100	98,784	100%	PCM
EnergyBullet	725,004	PCM	100	725,004	100%	PCM
PlayerMaterialize	1,323,000	PCM	100	1,323,000	100%	PCM
PlayerGlitch	88,200	PCM	100	88,200	100%	PCM
PreDetonate	1,506,456	PCM	100	1,506,456	100%	PCM
UBDetonate	352,800	PCM	100	352,800	100%	PCM
EnergySwirl	954,324	PCM	100	954,324	100%	PCM
ImpactCounter	95,256	PCM	100	95,256	100%	PCM
ImpactEvade	142,884	PCM	100	142,884	100%	PCM
ImpactBlock	144,648	PCM	100	144,648	100%	PCM
ScreenGlitch	88,200	PCM	100	88,200	100%	PCM
BugIdle	165,816	PCM	100	165,816	100%	PCM
HealthGain	400,428	PCM	100	400,428	100%	PCM
HealthPickUp	160,524	PCM	100	160,524	100%	PCM
EnergySeep	243,432	PCM	100	243,432	100%	PCM
SpiderLeap	144,648	PCM	100	144,648	100%	PCM
SpiderSwipe	93,492	PCM	100	93,492	100%	PCM
EnergyShine	42,336	PCM	100	42,336	100%	PCM
EnemyDodge	144,648	PCM	100	144,648	100%	PCM
BitBucksBounce	301,644	PCM	100	301,644	100%	PCM
BitBucksPickUp	264,600	PCM	100	264,600	100%	PCM
CircleSwitch	136,620	PCM	100	136,620	100%	PCM
CorruptedObjectGlitch	79,380	PCM	100	79,380	100%	PCM
InfoPageDesc	31,752	PCM	100	31,752	100%	PCM
Land	132,300	PCM	100	132,300	100%	PCM
PlayerFootsteps1	56,888	PCM	100	56,888	100%	PCM

Levels (Sound Bank)

Sound Name	Category	Priority	Options	Notes
PlayerDigitize	Player	0		
Explosion	ParticleEffects	0		
Fire	ParticleEffects	0		
Digitize	Default	0		
Materialize	Default	0		
SpinningBlade	Player	0		
PlayerReaperAttackPrimary	Player	0		
Jump	Player	0		
DoubleJump	Player	0		
HUDHealthBarGrow	HUD	0		
FirewallFizzle	ParticleEffects	0		
LandHard	Player	0		
BlockSpin	Player	0		
WallSlide	Player	0		
WeaponStick	Player	0		
DigitalCircles	Default	0		
ImpactHit	Default	0		
EnemySpawn	Enemies	0		
EnemyDespawn	Enemies	0		
VirusReplicateGlow	Enemies	0		
Spark	ParticleEffects	0		
PlayerReaperFinisherPrimary	Player	0		

Cue Name	Notes
PlayerFootstepWalk	
PlayerMaterialize	
PlayerDigitize	
Fire	
Explosion	
BlockDigitize	
BombDigitize	
BlockMaterialize	
BombMaterialize	
UBlockDigitize	
UBlockMaterialize	
PlayerAttackWaitSpin	
PlayerReaper1_1G_Primary	
PlayerReaper2_1G_Primary	
PlayerJump	
PlayerFootstepLand	
PlayerDoubleJump	

Figure 122: Screenshot of the XACT interface, showing the collection of wav files and the list of sound cues

Overall the user interface was fairly simple to use. All Dan had to do was insert the sound files into the engine, and then link them to sound cue names that Eric's code could recognize. Once the sounds were in the engine, Dan could manipulate different XACT variables to further alter the sounds. For example, footstep sounds could have their volume and pitch randomized, so that a slightly altered sound would be played each time the player character walked. This allowed for more realistic footstep sounds. XACT could also keep track of the distance a player was to a sound source, allowing for realistic attenuation. This ensured sounds would only play if the player was somewhat near where the sound source was in game, as opposed to all of the time.

Sounds that were supposed to be looped continuously simply needed to have a checkbox flagged, telling the engine to repeat the sound over and over again. If a sound was too overbearing in the game, there were volume modifiers that brought the sound down so that it would blend better with the rest of the sounds. The advantage of this was that Dan did not have to reimport a new .wav file with a softer volume, as the engine could take care of that for him.

As more and more sounds were added to the engine, it became increasingly more difficult to keep track of them. This stemmed from an initial lack of a standardized naming convention for audio files. Dan tended to name the sounds based on onomatopoeia, or what the files sounded like. This was a huge mistake, as it started to become unclear which files were linked to which sound cues, which would make future editing quite difficult.

Eventually Dan went back and renamed all of his sound files. He chose to mirror the names of them to match more closely to the sound cues that they would be linked to. Another thing that Dan failed to notice when learning how to use XACT was that there was a way to categorize the sound bank. By grouping different sound cues by categories, such as the HUD, and particle effects, it would become much easier to sort through the list of sounds as it became longer.

Obviously all of these organization methods should have been done from the very beginning, so having to stop and reorganize was an unfortunate, albeit necessary waste of time. Thankfully, by setting up the naming standards early on, it became a lot easier to keep track of the sounds in the engine the further Dan progressed.

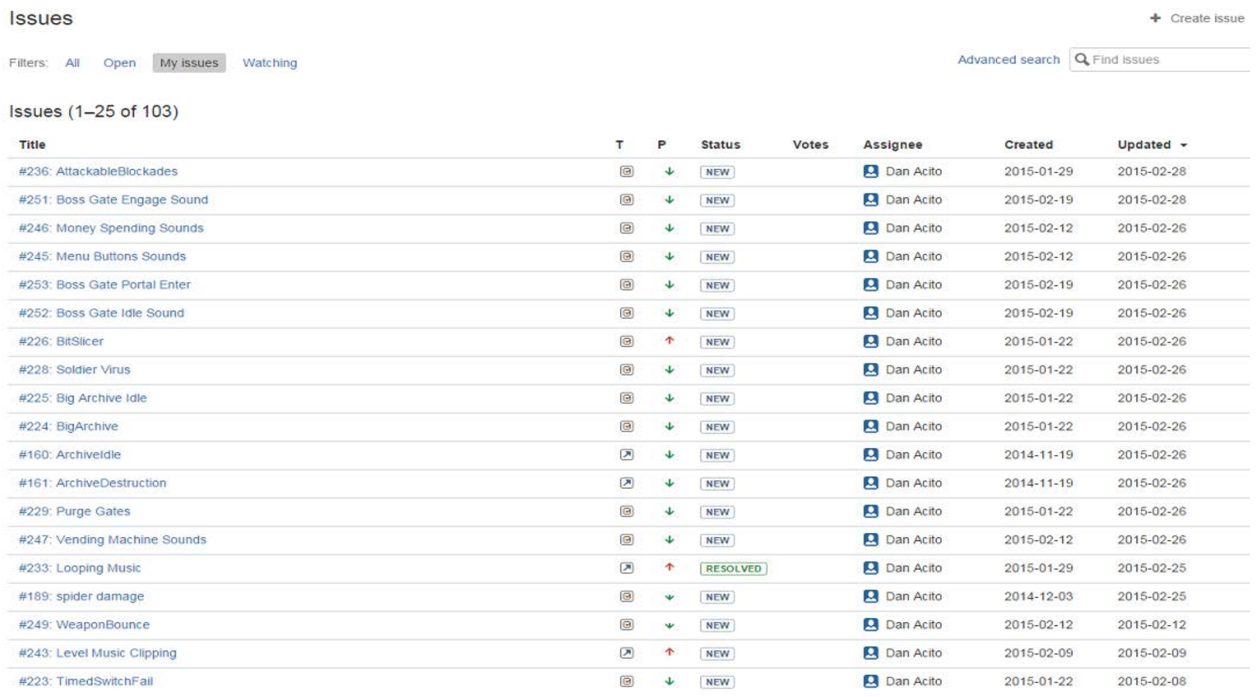
In order to test the sounds in the game itself, Eric gave Dan access to his level editor. Using this editor, Dan could quickly put together mini levels featuring the sound sources that he needed to look at or listen for. It was much quicker to have one small level with an enemy for testing sounds, than have to traverse one of Eric’s actual levels until Dan found an enemy. The advantages of this method was that there was quick turnaround. If a sound was supposed to loop and wasn’t, Dan simply needed to flag the sound cue in XACT. If a sound was much too quiet, Dan could go back to the engine and increase the volume of the sound.

In general, actual testing sessions were fairly short. Dan would listen to any of the new sounds that he had recently added to the engine, and see how they meshed with the rest of the environment. It quickly became clear whether or not a sound fit well or not. Some sounds simply did not work well in their intended use. Others were not timed well to an animation. In these instances, Dan needed to change the source sound file entirely, whether he needed to shorten or lengthen the sound, or simply replace it altogether.

In other instances, it worked as bug testing for Eric, as sometimes the sound cue was not hooked into the code properly. If no sound played at all, it meant that there was an error somewhere in the code that Eric would need to address. Another advantage of giving Dan access to the levels was that he would sometimes find additional bugs completely unrelated to the audio. It allowed the two to identify possible issues early in the development stage.

Once the majority of the sound effects from the asset list had been incorporated into the sound engine, Eric and Dan needed to go down the list and discuss which sounds needed to be improved. Many of the sounds worked well in the game, and Eric felt that they didn’t need much altering, while others were in dire need of improvement. The pair went down the asset list, listening to each sound file individually. If the two agreed that the sound was fine the way it is, they would mark the sound as complete on the asset list. If either thought that a sound needed to be improved, Dan would make a note of it.

The tasks of updating sounds were kept track of on an issue tracker through Bitbucket. Using the issue tracker, we could flag different tasks with priority on how soon they should be addressed. When Dan would begin to work on the final versions of the sound effects, he could start with the most important tasks first. This would include sounds that were more commonly heard throughout gameplay, or sounds that needed to be completely different than the original temp sound.


Issues

Create issue

Filters: All Open My issues Watching

Advanced search Find issues

Issues (1–25 of 103)

Title	T	P	Status	Votes	Assignee	Created	Updated
#236: AttackableBlockades	⊡	↓	NEW		Dan Acito	2015-01-29	2015-02-28
#251: Boss Gate Engage Sound	⊡	↓	NEW		Dan Acito	2015-02-19	2015-02-28
#246: Money Spending Sounds	⊡	↓	NEW		Dan Acito	2015-02-12	2015-02-26
#245: Menu Buttons Sounds	⊡	↓	NEW		Dan Acito	2015-02-12	2015-02-26
#253: Boss Gate Portal Enter	⊡	↓	NEW		Dan Acito	2015-02-19	2015-02-26
#252: Boss Gate Idle Sound	⊡	↓	NEW		Dan Acito	2015-02-19	2015-02-26
#226: BitSlicer	⊡	↑	NEW		Dan Acito	2015-01-22	2015-02-26
#228: Soldier Virus	⊡	↓	NEW		Dan Acito	2015-01-22	2015-02-26
#225: Big Archive Idle	⊡	↓	NEW		Dan Acito	2015-01-22	2015-02-26
#224: BigArchive	⊡	↓	NEW		Dan Acito	2015-01-22	2015-02-26
#160: ArchiveIdle	↗	↓	NEW		Dan Acito	2014-11-19	2015-02-26
#161: ArchiveDestruction	↗	↓	NEW		Dan Acito	2014-11-19	2015-02-26
#229: Purge Gates	⊡	↓	NEW		Dan Acito	2015-01-22	2015-02-26
#247: Vending Machine Sounds	⊡	↓	NEW		Dan Acito	2015-02-12	2015-02-26
#233: Looping Music	↗	↑	RESOLVED		Dan Acito	2015-01-29	2015-02-25
#189: spider damage	⊡	↓	NEW		Dan Acito	2014-12-03	2015-02-25
#249: WeaponBounce	⊡	↓	NEW		Dan Acito	2015-02-12	2015-02-12
#243: Level Music Clipping	↗	↑	NEW		Dan Acito	2015-02-09	2015-02-09
#223: TimedSwitchFail	⊡	↓	NEW		Dan Acito	2015-01-22	2015-02-08

Figure 123: Screenshot of Bitbucket Issue Tracker, listing tasks that Dan needs to address

For many of the polished sounds, it was simply a matter of adjusting volumes or editing the sound file so that it fit better with an animation. Other sounds needed to be completely redone. As before, Dan would find some sort of source sound to use as a starting point, and then would proceed to edit it using Reaper. When Dan thought that the sound was finished, he would audition it to Eric for approval. Once they both agreed on the sound, Dan would mark it off as completed on the asset list.

The last step, once all of the sounds were in the engine, was to do a final pass through each file in XACT. Once all of the music was introduced (see section on music), this would ensure that all of the sounds were perfectly blended together.

Editing and integrating the sound effects into the game was certainly a lengthy task, but Dan's goals were all met. *Xeero* was now full of audio cues that dramatically improved the quality of the game, adding to the sense of immersion and providing more direct feedback. As Eric continues to add more content to the game, this process will repeat itself as new objects and animations will need new sounds to accompany them; furthermore, there will always be room for improvement as Dan continues to hone his audio editing skills.

5.2. Music

The following section will look at Dan's second job for this project, which was to write and record the music that would serve as the soundtrack for *Xeero.* We will discuss the steps that Dan took to prepare for this task, including obtaining the appropriate hardware and software. Next we will discuss how Dan came up with the concepts for the music, as well as his actual development process. Finally we look into how the music was integrated into *Xeero*, and the different obstacles Dan faced along the way.

In terms of software, Dan was most familiar with Logic Pro for music production. Having taken several courses during his undergraduate studies relating to music using Logic, it was easily the best choice. Dan did not have Logic at the time, but was willing to invest in the software. It was the 10th iteration of the program, where Dan was most familiar with earlier versions; therefore, it took Dan several days to become accustomed to the program.

In terms of how Dan would orchestrate the music, the best option was for Dan to make it himself. Logic provided a lot of unique virtual instruments, many of which would be ideal for the digital domain that *Xeero* took place in. Logic has a feature that mapped Dan's laptop keys to that of a musical keyboard. The problem was that this did not quite feel right to Dan, and it was also incredibly inefficient.

The musical typing functionality was impractical as it overlaid the rest of the audio workstation, so often information that Dan would need to look at would be misconstrued. Clearly the best option for Dan would be to use an external piece of equipment.

One thing that Dan had access to was his old keyboard controller for the musical game *Rock Band.* This controller has a MIDI output port on the side that enables the controller to be used as a MIDI keyboard. While this was certainly not the most advanced of MIDI controllers, it did have a feel that was more similar to that of a piano. This would make it much easier for Dan to perform music, and was a more comfortable fit.

Dan simply needed a MIDI cable that could connect this controller to his laptop. Not having a cable handy, Dan ordered one to be delivered to his apartment. When the cable arrived however, it did not function properly. Dan's computer could not recognize the MIDI device.

Dan decided to seek Keith Zizza again for help and recommendations. Zizza could not diagnose why the cable Dan had obtained did not work, but he did offer Dan another solution. Zizza loaned Dan a couple of his own MIDI cables, as well an external audio interface that would allow Dan to connect his keyboard through his laptop's USB port. This would greatly aid in the mixing quality and improve the sound of the music. Now that Dan had all the materials he needed to get started, he was now ready to begin writing and recording.

Dan had a lot of ideas for what he wanted to do with the music, but he wanted Eric's input first before Dan would actually start the work. Dan asked Eric to provide him some sample songs that were of a similar style to what Eric wanted for *Xeero*. Eric gathered a few songs from some of his favorite video games and sent them to Dan. These works included composers Yoko Shimomura for her work on the *Kingdom Hearts* series, and Carlo Castellano for the game *Invaders: Corruption.*

Shimomura composed her songs as orchestral arrangements designed to be intense and suspenseful. Castellano's music was more digitally oriented, but was arranged to be upbeat fast-paced. Combining these two styles would be the approach that Dan would take for creating the music for *Xeero.*

Eric and Dan determined that roughly 4 to 5 songs were necessary for the first installment of *Xeero*. There would need to be a theme for the actual levels, enemy combat, the final boss, the overarching world hub, and one for the title menu. Generally the tunes were to be fairly upbeat and action-like. Eric professed a liking for classical instrumentation, with a blend of more digital sounding instruments. This is the direction that Dan would take when arranging the different pieces of music for *Xeero.*

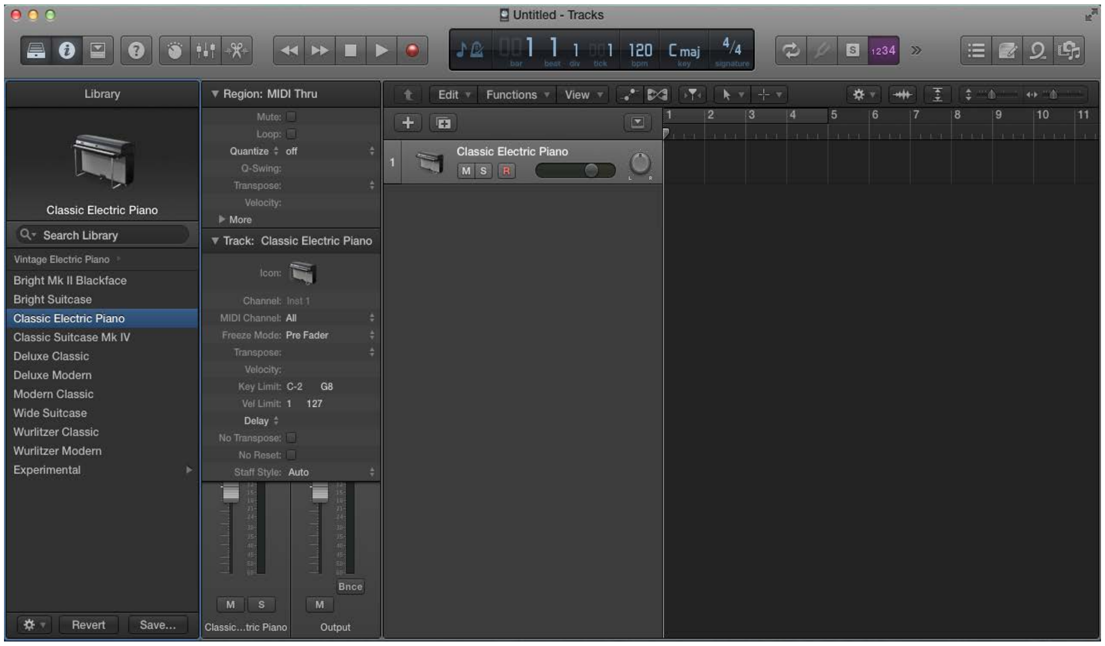
When Dan began to work on the first music piece, he did not really have a use for it in mind. His ultimate goal of this piece was to make sure he was completely re-familiarized with the Logic interface, as well as to experiment with the different virtual software instruments in Logic. This first session would help Dan to find multiple instruments for the different layers that each song would have.

In terms of the development process, Dan liked to start with one track, and then add on more layers as he went. Typically he would start with either the bass, or the rhythm tracks. After finding an interesting sounding instrument, he would play around on the keyboard controller to see if he liked how the different notes or beats flowed together. Many drum kits had lots of different percussion instruments to choose from. A bass drum would be mapped to the low C note on a keyboard, and then the keys would progress to the snares, cymbals, and toms.

Typically all of the songs Dan wrote followed a standard ABA format. Under this arrangement, the ‘A’ was one section of the music, the ‘B’ was a second section, typically a bridge, and the second A was a variation of the first section. This kept the songs very simple, but not too repetitive.

The easiest way for Dan to sample the instruments was to make a short 4 or 8 bar set, and then make the section loop seamlessly. If the drum beat flowed well, this loop could be utilized for most of the song. As more layers were added, Dan would add some variance to the beat and add some drum fills to the different verse transitions.

Once a stable drum beat was set, the next layer that Dan would work on would be the bass. This would set up the chord progression that Dan wanted for the song, and help keep a consistent timing for when the melody was to be added. Bass was slightly trickier than the drums because the keys were a lot more sensitive to velocity, or how fast the key was pressed, for stringed instruments.



Region: MIDI Thru
Quantize: off
More
Track: Classic Electric Piano
Channel: Inst 1
MIDI Channel: All
Freeze Mode: Pre Fader
Key Limit: C-2 G8
Vel Limit: 1 127
Staff Style: Auto

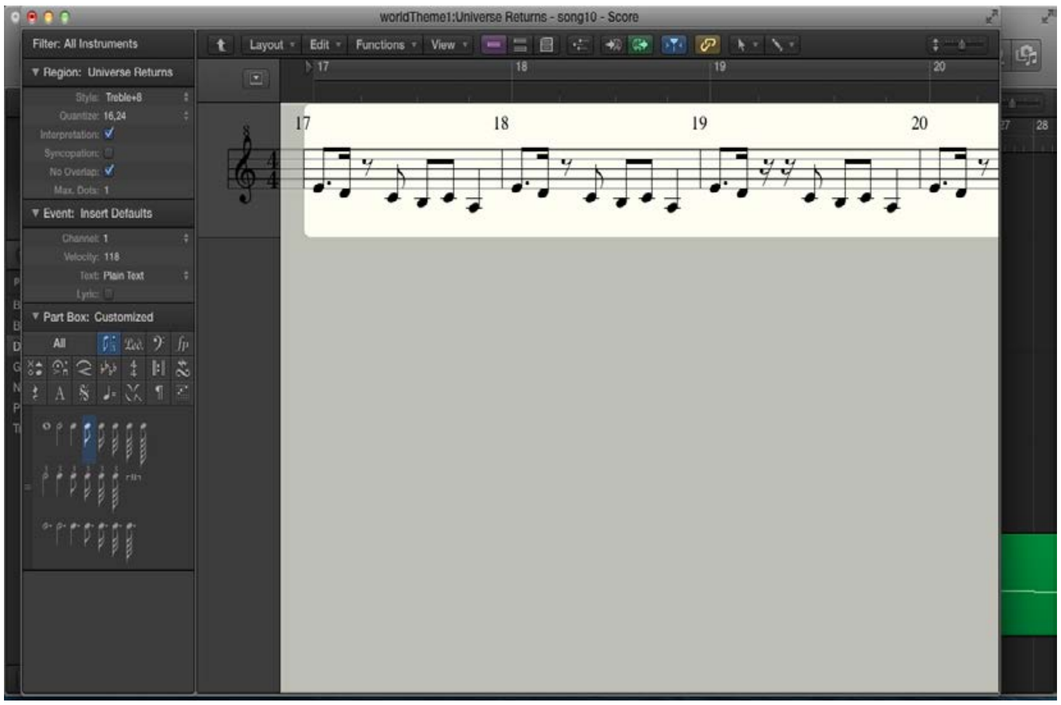
120 bpm, C maj, 4/4

Figure 124: Screenshot of the Logic Pro X Interface

It would often take several takes before Dan could get the notes to all sound similar. Writing and performing the melody was always the most difficult part for Dan when it came to arranging the music. The melodies did not have to be too complex, as Dan did not want to music to take too much of the player’s attention. Dan would write the melodies to be simple enough to loop, but interesting enough so that one could enjoy listening to it.

One of the biggest issues of using MIDI was that the notes performed would not match well to an actual music score. Dan would often use the score editor function in Logic to tweak minor changes to the music. This would include changing an individual note’s pitch, deleting a note, or even duplicating a note. The hassle with this process is that musically, the notes would sometimes not play on the correct beat, despite the note being correct on the music staff.

Generally, all notes on the staff in the score editor would play around a ⅛ of a measure late. To compensate for this, Dan would have to consider this delay any time he needed to make minor adjustments using the score editor. While the process worked, it was rather tedious and inefficient.



Filter: All Instruments

Region: Universe Returns

Style: Treble+8
Quantize: 16,24
Interpretation: ✓
Syncopation: ☐
No Overlap: ✓
Max. Dots: 1

Event: Insert Defaults

Channel: 1
Velocity: 118
Text: Plain Text
Lyric: ☐

Part Box: Customized

Figure 125: Screenshot of the Score Editor function in Logic Pro X

Once the main melody was in place, Dan would work on the bridge, or 'B' section. Generally this section had its own unique melody, and would often remove some of the other voices from the previous verse of the music. Typically there would be no percussion, and the music would be somewhat soft. Dan's motivation was to use this bridge as build up to the next section, which would be a variation of the main melody.

The next verse would sound very similar to the first. Typically Dan would copy the music data to the new measure and logic and then play around with the notes to add some variation. It could be something simple such as adding or removing a few notes, or changing the octave that the music was playing in. The overall structure and chord progressions would stay the same. The idea was not to change too much, but alter the melody slightly to keep the music interesting.

Once the basic structure of a piece was set, Dan would work on adding additional textures and layers. These could include occasional ambient noise, various chords, or just interesting instrumentation that added more depth to the music. The additional textures helped to round out the sound so that the music was much fuller.

The ambient sound also would fit well with the in-game environment, since the setting is often mysterious and ominous.

The last step of the music arranging was to go through the mastering process. This would entail making sure each of the track layers blended well with one another. One way that this was accomplished was through volume automation. Use the automation tool in Logic, the volume for each layer could seamless be altered throughout the whole song.

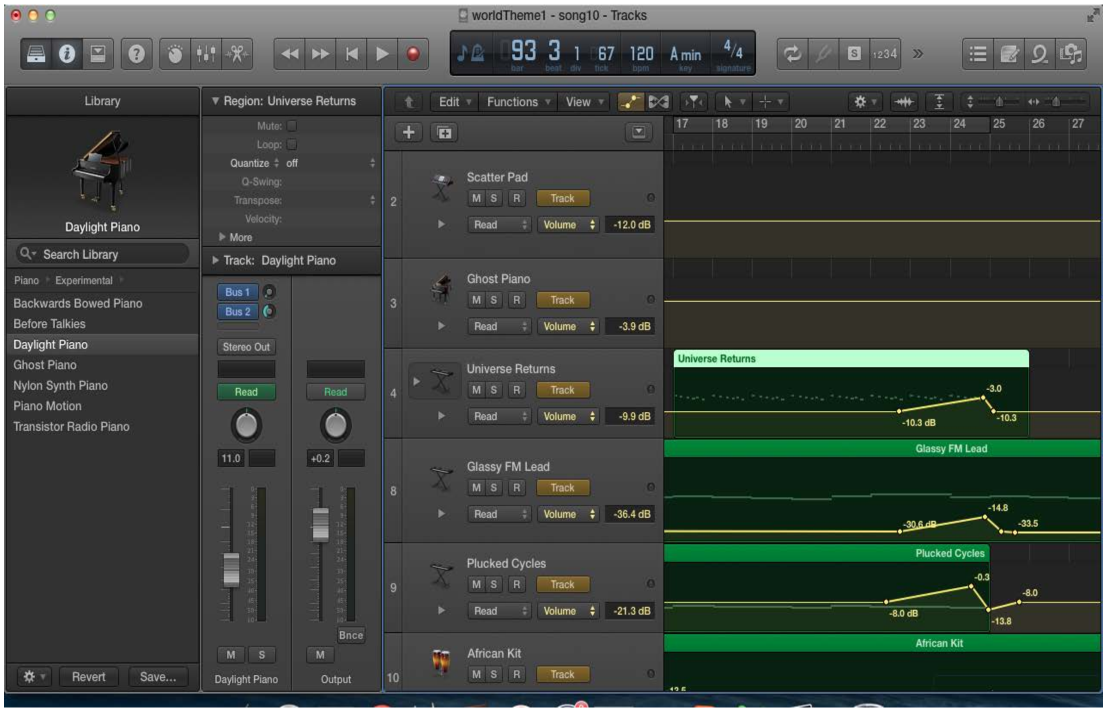


Figure 126: Screenshot of the volume automation function in Logic Pro X.

This would ensure less important layers would be brought down in volume so that the more important ones could be heard, and vice versa. It also made it much easier to have more seamless music transitions through crossfading.

Lastly the different layers would be quantized. This would restrict the notes to a more rigid format, and automatically correct the timing of the different audio regions. While Dan timed the notes the best he could, the quantization process ensured that each region would be timed perfectly.

Eric told Dan that due to how the engine streamed audio from disk, the audio would not initially need to be compressed. This allowed Dan to render the music in .wav format, so that it would have a much higher quality. This would also ensure continuity with all audio in the game and the audio engine.

Once the music tracks were mastered and rendered, Dan would need to add these files to the audio engine. A separate sound bank was created specifically for music to stream. This would help so that the large files would not take up too much computer memory during gameplay. As with the sound effects, the music was linked to a sound cue that Eric's code would recognize.

Once the music had been inserted into the audio engine, it was clear that some things would need to be adjusted. The biggest and perhaps most noticeable issue was that the tracks were not designed to loop. Each song had a unique intro and outro which did not seamlessly connect if the song was to repeat itself. This would create a rather jarring jump in the music, and could easily break player immersion.

To make up for this, Dan would need to go back into Logic and repurpose the music to better fit into the game. The music files would be cut into three different sections: an introduction, an ending, and a mid-section. This mid-section would be arranged so that it would seamlessly loop. The introduction would play once at the beginning of the stream, and from there, the mid-section would play through the rest of gameplay. The outro would not be heard in game.

The last step was making sure that all of the sounds and the music blended well together. If there were any overlapping frequencies, the sound would become distorted and jarring. Thankfully, for the most part, there weren’t too many frequencies that needed adjusting. The main thing to do was to balance the volume between the audio cues and the music stream. The music was actually much quieter than the audio cues, so Dan needed to increase the volume on the music stream slightly. Once this was accomplished, the game audio was complete.

The music development task was very long, but certainly very rewarding. Dan was able to arrange several pieces of music that fit quite well into the game. The process certainly helped Dan to improve his musical abilities, as well as gave him a perfect chance to practice his editing techniques. Future levels of the game would ideally have their own unique music, so Dan would need to repeat this process again for any future development.

6. Technical Development

Eric was responsible for the technical development of *Xeero*. The following section describes the technology used in the development of the game and some of the major systems created in development.

6.1. Framework and Platforms

Xeero is built with Microsoft XNA Framework, specifically for Windows. The game, as mentioned in the **Concept Origin** section, was originally built as a final project for an XNA game programming class, making the choice of framework a necessity for this project.

Microsoft XNA was built to release games on various Windows platforms: Windows PCs, Xbox 360, and Windows Phone 7. However, as of April 1, 2014, Microsoft officially discontinued the platform (Hruska). While the framework can still be used develop games for Windows using DirectX 9, XNA cannot specifically target later generations of Windows products.

However, the library MonoGame, originally created by José Antonio Leal de Farias, was developed as an open source implementation of the XNA framework (Stolpe). As of March 2015, MonoGame extends the platforms supported by XNA, including Windows 8.1, Mac OS, Linux, iOS, Android, Windows Phone 8, and Ouya, as well as major consoles like Microsoft Xbox One and Sony PlayStation 4 (Jackson). MonoGame and XNA framework share a nearly identical API, designed to allow developers to easily port XNA games into MonoGame.

The game is designed for Windows, with gamepad support. Through the underlying architecture of the game, it can be easily ported to the Xbox 360, and releases for Linux, Mac, PlayStation 4, and the Xbox One are possible with relatively few changes to the codebase.

Xeero is designed to be released initially for Windows PCs, but will switch to MonoGame to target multiple platforms.

6.2. Building the Engine

Microsoft XNA is a framework that serves as an analog to their DirectX API, and contains functionality to render 2D and 3D graphics, play audio, and load game assets. XNA is designed to be used with the Microsoft .NET language C#. Many of the features discussed in this section use the C# implementation of object-oriented programming, like inheritance and polymorphism through classes and interfaces.

Eric developed a custom game engine to run *Xeero* using the underlying features provided by XNA. The engine provides a renderer, a physics engine, a particle system, an audio player, an asynchronous content manager, GUI management, animation, AI, and a level editor.

The following sections describe some of the major systems and notable features implemented for *Xeero*.

6.2.1. WorldObjects

The foundation of the engine is built around "WorldObjects." These represent *every* element of a level, from the player and enemies to level geometry and invisible triggers. These provide generic functionality that can be specialized by creating derived classes.

WorldObjects provide basic information about an object. At its core, every WorldObject is a physics object and has a transform, defining its position and orientation in space, and a collision volume, which determines how much space in the world the object takes. Each WorldObject also contains inheritable functions that are automatically called over the lifecycle of an object in a level, letting the programmer process the object when it is added to or removed from a level, when it is supposed to draw itself, update its physics, or respond to collisions.

WorldObjects also allow the designer of levels to define relationships between these objects. Each object supports “linking” to another object, to denote that these objects are related. Based on the type of the object and the behaviors it implements (see **Behaviors**), these relationships can behave differently. Most commonly, these links can be used to denote “switch/receiver” relationships to, for example, allow a **Button** to be linked to a **Gate** object, allowing the gate to be opened when the player stands on the button. Links are directional relationships (so a button linked to a gate has different meaning than a gate linked to a button), and any WorldObject can be linked to any arbitrary number of other objects, allowing complex WorldObject interactions. Child WorldObject classes can also restrict the types of objects to which they can be linked, disallowing the designer of levels from linking arbitrary objects whose relationships are undefined.

WorldObjects allow most of their customization through inheritance; the program can derive a class from WorldObjects, implement specific functionality to run over the lifecycle of the object, and add those objects to levels. Additionally, WorldObjects allow **Behaviors, Interfaces,** and **Animators** to be added to or implemented through them to give the objects preset functionality and to interact with other WorldObjects in different ways.

Behaviors

Behaviors are the most flexible elements that can be added to WorldObjects; these discrete objects can entirely define the functionality of an object. WorldObjects of different types allow these behaviors to be added to them to give them common properties. Behaviors have access to all of the lifecycle methods of WorldObjects and can force the WorldObject to react to different events in different ways.

For example, a FirewallFizzler behavior, when hooked to a WorldObject, will automatically destroy the object and release a particle effect when the WorldObject touches the firewall at the bottom of every level. This is a behavior that is very common among WorldObjects, but isn't specific to any specific game element. This allows objects of different types, like **Enemies** and **Archives**, to have the same, standardized behavior.

Behaviors can be used to define gameplay interactions between objects, like the BrokenFaller behavior that causes an object to shudder and fall when the player stands on it, or they can provide engine-level functionality, like the ModelRenderer behavior that allows the object to set a model to be drawn on the screen.

Behaviors are designed to implement standardized actions among WorldObjects and reduce the amount of redundant code that would otherwise need to be written. Behaviors answer the question of *how* a WorldObject behaves.

Interfaces

Alternatively, interfaces determine *what* a WorldObject does. WorldObjects use interfaces, the object-oriented programming concept, to understand different types of objects and to understand how different objects are allowed to interact.

For example, objects that implement the IHitTarget interface advertise that these object can be struck by weapons. When the player is attacking, if their weapon collides with an IHitTarget object, the weapon then gives information to the object about the type of hit, like the position of impact, the source of the attack, and how much damage the attack would do, without any regard to *how* the hit target would respond to the attack. Most interaction between objects therefore do not occur at the per-object level, instead checking against specific interfaces. This allows objects like the **Compressor** to detect motion of *all* attackable objects, rather than just the player, to allow the player to attempt to use the compressor in combat to defeat enemies, expressing one of the "moments" of gameplay *Xeero* attempts to create the "combined interaction of game mechanics."

Animators

While behaviors give WorldObjects specific reactions to different events and interfaces give WorldObjects a generalized way to interact with each other, animators are used *within* WorldObjects to create standardized animated motion across different objects as a result of different stimuli. For example, a ShakeAnimator, when triggered, will “vibrate” an object for a short amount of time. Some objects that implement the IHitTarget interface will use a ShakeAnimator to vibrate themselves in a standardized way when they are struck. Many objects, like **BitBucks** and **Health Data** that populate levels, float slowly up and down while they are in a level, and, as such, use the functionality provided by FloatAnimators.

6.2.2. GameWorlds

WorldObjects are grouped together through GameWorlds. These are the levels that create the hubs and programs. GameWorlds can be saved to and loaded from files, to facilitate level loading for the level editor (see section **Level Editor**). GameWorlds save each WorldObject that inhabits the world and the links between them. GameWorlds also manage physics and rendering of the WorldObjects.

Physics Engine

Originally, Eric built a custom physics system to work with *Xeero*. This allowed for basic control over objects using dynamics and collision detection. While functional, the system was problematic.

The system could handle and resolve collision between basic collision shapes, like axis-aligned bounding boxes and bounding spheres, but the system could not resolve oriented bounding boxes, and thus entirely ignored orientation, rotational velocity, and rotational acceleration. The system also used rudimentary grid-based space partitioning, and suffered from severe performance dips when relatively few objects existed in proximity.

Mid-way through development, Eric researched third-party physics engines to integrate into the game that satisfied the constraints of the project: the engine needed to be high performance and designed for real-time applications, free and open licensed, and built for the Microsoft .Net platform.

After investigating engines like Havok, Jitter, Farseer, and BepuPhysics, Bullet Physics was chosen. Bullet Physics, a high performance physics engine, satisfied most of the requirements of the project, but the engine was originally built to be used with unmanaged C++. However, programmer Andres Traks created BulletSharp, a wrapper for the Bullet Physics library. Not only was BulletSharp designed to work both with the .NET platform, but it also contains separate builds to integrate the API with XNA, MonoGame, and other game development frameworks.

Bullet Physics was integrated into *Xeero*'s engine using BulletSharp, and the source code of Bullet Physics was modified to allow the physics engine to cooperate with the specific requirements of the game.

Rendering Engine

The rendering engine, while it has undergone several iterations and improvements, did not incur as dramatic a redesign as the physics engine did.

The bulk of rendering time in *Xeero* is devoted to rendering hundreds of discrete models used in the modular elements that make up GameWorlds, so the rendering engine was designed to optimize rendering performance. The engine optimizes rendering in two major ways: hardware instancing and object culling.

In any given level, the majority of models used in the level are repeated; for example, the **Platform** is a standard building block of a level, and a level could contain hundreds of them. *Xeero* leverages DirectX 9's hardware instancing implementation to reduce the overhead of drawing hundreds of discrete instances. Using hardware instancing, a model is stored once in buffers in video memory, and all of the instances that use that model are also stored in a buffer on the GPU, requiring the engine only make a single call to the GPU to draw multiple instances of an object. Using this technique, most of the objects in the game are instanced, reducing the bottleneck of communication between the CPU and GPU.

Additionally, the engine attempts to cull as many full objects from the scene to be rendered as it can. Levels in *Xeero*, being large, open, and reminiscent of a void (see **Visual Art - Environment**), allow the players to see most of the objects that are in the level. This can cause high overhead for the GPU to render every object in a level each frame. So, the engine attempts to cull as many instances of objects as it can before calling the GPU. Most objects are culled using frustum culling: objects outside the camera's frustum cannot be seen and are therefore not drawn. Other objects can be culled based on their visibility and transparency settings.

6.2.3. Content Management

Early in *Xeero*'s development, loading game assets was handled simply: every asset in the game was loaded all at once before the game started, and were used as needed. Working under these assumptions, content in-game was always considered present when it was needed, and the code was written accordingly.

Predictably, this system became problematic; when the number of assets needed for the game grew, load times grew longer, resulting in delays of nearly a minute before the game would start. Using so many assets had a high memory footprint, running the risk of the game running out of memory before the application even started.

Thus, an improved content system was developed. This system allowed the code to treat game assets as if they were constantly present and allowed the game to "lazily" load content, or only load game content as it was needed.

Content in the game was modified, so rather than directly handle assets, WorldObjects handled lightweight “tags” to assets. These WorldObjects could treat the tags as the assets themselves, and these tags would only be translated into assets when the asset was needed for rendering; if the asset wasn’t loaded, the renderer wouldn’t attempt to draw the asset.

Each tag, when used, would send a notification to the content loading system that an asset was requested, and a separate thread would load the asset while the game runs. This system also allowed for flexible control over groups of assets. Each asset would be tagged for its usage, from general purposes like “game levels” or “splash screen,” to more specific object tags like “block objects,” or “compressor objects.” The system allows groups of assets to be loaded or unloaded based on those tags. Assets could also be given priority, so when a cluster of assets are required at once, the content loading thread would prioritize and load the more important assets first, so the model for a critical gameplay object would be loaded before an asset for a supplementary visual effect.

This system allowed for any number of assets to be loaded before the game starts, and the remainder of the assets are only loaded as they are needed. Asset loading can also be modified by the programmer, to load only a partial list of assets during loading screens to help reduce loading times.

6.2.4. Animation Controllers

As mentioned in the **Animation** section of **Visual Art - Style - 3D** chapter, character animations needed to smoothly transition from one to another to facilitate the player quickly performing a series of actions in succession. To both reduce the number of transitional animation that would be required to make and allow animations to be interrupted, an animation blending system was developed. The animation controller is a “behavior” the programmer can hook into a WorldObject.

This system gives the programmer flexibility over which animations are played for a single animated model. A single animation is considered a “blend state” for the model. The model can play a single blend state, but, before the animation is completed, a new animation can start. This animation is allowed to “blend” together, to transition from fully using the old animation to fully using the new animation over time. This system allows blend states to transition from one to another, but also allows any arbitrary number of animations to be blending at one time. For example, if the player starts to run, the run animation begins to blend from the idle animation. But before the animation completely blends, the player can press the jump button, causing the jump animation to start playing. In this case, while the animation is blending from “idle” to “running,” the resultant pose is then blended from that into the “jump” animation. If the player then decides to attack before those animations have finished blending, then *those* resultant poses will be blended into the “attack” animation.

The animation system also allowed more complex animation combinations through “layers.” Each layer contains a stack of blend states blending as described above. But each layer can also be blended together in a similar manner. And these layers can also be triggered to only manipulate certain bones on the model. For example, the player could be blocking, and the blocking animation is playing. If the player is in mid-air, and suddenly lands on the ground, a new animation layer is created only for the legs of the player. This layer then plays the landing animation, while the player’s arms are still following the blocking animation.

Even further, the programmer has more precise control over specific animations used in blend states. The programmer can determine how the animation loops, how far into the animation to start playing, and even edit the motion of bone in the animation, allowing, for example, the Xeero to move his arm and head to point at the object he is digitizing, even if the object is above or below the preset location of the arm and head in the animation.

This control created a robust, flexible animation system that could be used to control the player, enemies, and even various props.

6.2.5. Lua Scripting

The engine also allowed moderate control over the object within levels through Lua scripting. A Lua engine is instantiated and can be controlled through commands issued with the console that opens in debug versions of the game's executable. The scripting commands allow the creation and destruction of WorldObjects within levels, moving objects, editing save data, and changing game settings, all while the game is running.

6.2.6. Play Recording

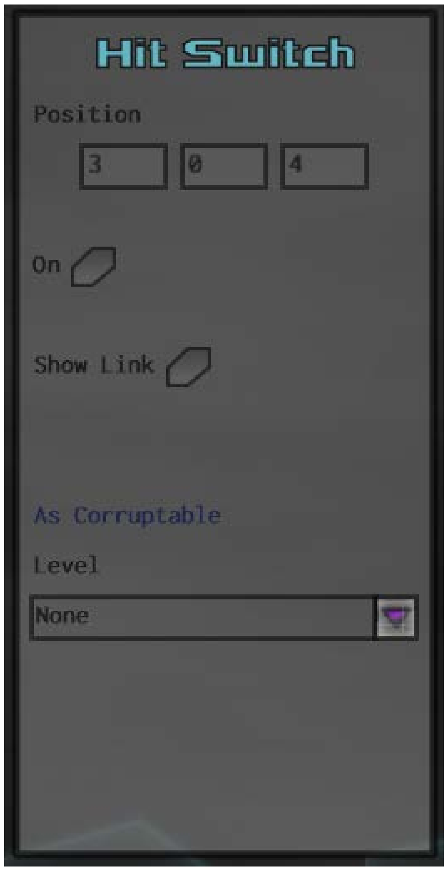
The engine also allows for lightweight play session recording. The game can capture data as players play the game and save the data into lightweight "move data" files. These files can be loaded into the engine, and the play session can be recreated, showing the position and actions of the player character and the movement of camera by the player as Xeero moves through levels.

Rather than saving video frame data, the move files store the GameWorld the move file is recording, the position and animations of the player character, the position of the camera as controlled by the player, and the position and states of any relevant object each frame. This data can then be stored compactly to be replayed when needed.

Play recording is a valuable tool when playtesting the game, allowing Eric to re-watch players playing the game and see how these players react to different sections of the levels. This also allows move data to be separated, and allows, for example, only the player character position and animation data to be extracted to show players a “ghost” of their previous play-throughs of a level while they play.

6.2.7. Level Editor

As mentioned in the **Design - Level Editor** section, a level editor, Programmer, was created to facilitate the creation of GameWorlds. This level editor needed to be powerful enough to allowed diverse levels to be created with dozens of different types of objects populating the level. Additionally, new objects and game elements were being built over the entire lifecycle of the project, and many of those objects were required to be present in the level editor. The objects themselves each had properties that the designer of levels can change. The level editor also needed to be flexible enough to not require any maintenance to the editor to add new objects or change the properties of those objects, so the focus could be given to creating new gameplay, not ensuring the level editor catches up with the new objects.


Hit Switch

Position

3 0 4

On

Show Link

As Corruptable

Level

None

Figure 127: A argument selector for Hit Switches, pulled directly from the constructor

To facilitate this, the level editor in *Xeero* relies heavily on reflection and letting the definition of WorldObjects themselves be the maintenance required to add

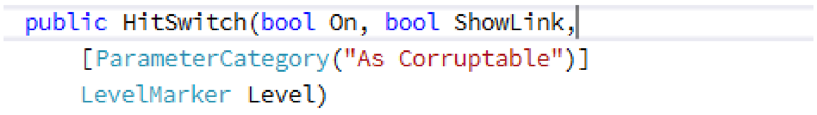

```
public HitSwitch(bool On, bool ShowLink,
    [ParameterCategory("As Corruptable")]
    LevelMarker Level)
```

Figure 128: The constructor for a HitSwitch, which is used in the argument selector in the level editor

them to the level editor. To determine which objects should be entered in the editor, the level editor looks for a specific “Editor Entry” C# attribute that is specified in the definition of the class. This attribute, which also contains a directory in which to store the object, tells the level editor to allow that type to be created from the editor. It also looks for an “Editor Constructor” attribute on a constructor for the class, so the editor can use the constructor for the object to directly construct the object from designer input. The editor understands basic data types like integers, floating-point numbers, strings, and enumerations, and is able to give the designer the appropriate GUI controls to allow them to enter the data for the object’s constructor.

One of the inheritable methods WorldObjects share are “editor metadata methods.” These allow WorldObjects to specify what kinds of data to use when displaying the object in the editor. The object can specify models, particles, and wireframe information to be rendered.

7. Project Management

Anthony was responsible for handling all things project management. This section will discuss every facet of the project management process for Xeero. Meetings, milestones, promotion, and any other tasks that were undertaken to move the project along will be dissected and talked about here.

7.1. Production

Since he came onto the project in the middle of the development process, Anthony had to alter his approach slightly compared to his MQP. With his senior project, Anthony had to budget time for the concept stage. Xeero was past that stage by the time he joined and development was well underway. As a result, Anthony was responsible for guiding the project through the alpha and beta stages. Since this was the case, more time could be worked into the schedule for festivals and playtesting. As one of the goals of this project is to eventually release it for public purchase, any extra time that can be spent testing it and putting in front of others is a huge advantage.

7.2. Scheduling/Team Meetings

In order to facilitate the completion of tasks and move the development of Xeero along, Anthony scheduled multiple meetings between team members of the course of development (see **Appendix D** for an example Gantt chart detailing work to accomplish over the semester). The following three meeting types were scheduled each week:

7.2.1. Team

In these meetings, the team reviewed the tasks that were accomplished over the past week and laid out the tasks that would be undertaken for the upcoming week. These meetings lasted anywhere from 30 to 60 minutes (see **Appendix E** for an example of minutes created for a meeting).

7.2.2. Prep

These quick meetings occurred just before Advisor meetings and provided the team the opportunity to sync up one last time before meeting with project advisors. These meetings typically lasted anywhere between 15 and 30 minutes.

7.2.3. Advisor

These meetings occurred each Monday with Professor Moriarty. Over the course of an hour, the team provided Professor Moriarty an update on the development of the project and what the team hoped to accomplish over the next week. Upcoming events such as open houses or festivals were discussed as well.

7.3. Festivals

Since Xeero is an indie title, getting public exposure is more difficult when compared to a AAA title. As such, festivals were a key resource that the development team took advantage of to get the game in front of players. Anthony was responsible for registering Xeero and the team for these festivals and coordinating the logistics for attendance. A spreadsheet was created and updated over the course of development that provided information on all of the upcoming festivals and conferences that the team wanted Xeero to be a part of. PAX East, Made in Mass, and Boston Mini-Fig were just a few of the festivals that Xeero made it to.

At the time of this writing, Xeero has been shown at the WPI booth at PAX East as one of the four student projects as well as at its own table at the annual Made in Mass party at the Microsoft NERD center. Gamers got a chance to try out the first couple of levels as well as provide suggestions and feedback on what they saw. Eric, Dan, and Anthony were on hand at both events to help facilitate the play sessions. Future festivals that the team plans on attending include E3, Boston FIG, IndieCade, and the RPI Game Fest (see **Appendix F** for a larger list).

7.4. Playtesting

As with any piece of software, adequate testing is required in order to ensure the game works properly and that it is enjoyable to players. Anthony was responsible for coordinating and setting up playtesting sessions for Xeero.

Playtesting in Xeero unfolded in two iterations. First, the tutorial level was demoed to a class of 25 game design students. These students played the tutorial, then answered a brief post-play survey Eric created (see **Appendix G**).

Then, as an assignment for a game production class in which Dan, Eric, and Anthony were enrolled, a more formal testing protocol was developed for the game (see **Appendix H**). This protocol involved running both an in-game and a post-play survey. While testers played the game, a test proctor would observe the session and ask players questions from the in-game survey (see **Appendix I**). Then, after the tester completed the suite of levels, they would fill out a post-play survey (see **Appendix J**).

With the help of playtesters, various bugs and glitches were discovered in the levels that had been completed and were promptly fixed. One example that stands out involves some of the doors towards the end of the second level. The doors are programmed to activate once a certain number of switches are triggered. On rare occasions, upon triggering the switches, the door geometry would become garbled and the door wouldn't open. This particular level had been played through multiple times by Eric, Dan, and Anthony on many occasions but the issue did not surface until the game was put in the hands of testers. With a game as large in scope as Xeero, no amount of playtesting is ever enough.

7.5. Trailers

In order to get people interested in Xeero, they have to be introduced to it and see it in action beforehand. Trailers accomplish just that. Anthony was responsible for creating multiple trailers over the course of development to show off at the various festivals and open houses that Xeero was a part of. Eric captured gameplay videos using Camtasia and Anthony edited the content together using iMovie.

At first, the early iterations of the trailer were too long. Originally, they clocked in at around 80 seconds and contained gameplay clips that were either too long, too repetitive or dull. Professor Moriarty provided feedback on each version of the trailer and it was tightened up and refined into a much more polished video. By the time the project was completed, the trailer had gone through four iterations based on the feedback and the availability of new gameplay videos. The video ended up with a final running time of just under one minute. Trimming the fat off the trailer so to speak really aided in creating an exciting video for gamers to watch.

8. Postmortem

In this section, we will reflect about our individual experiences with the project. We will discuss what worked well for us and the methods that we found were most successful. Additionally, we will discuss some of the issues that we ran into: what went wrong and poor decisions that we might have made. Finally, we will wrap up by discussing what we might have done differently and what we took away from this experience.

8.1. What Went Right

8.1.1. Eric Anumba

Testing and iterative refinement were a boon for the project; by developing, testing, and refining gameplay in *Xeero*, the team is able to create a game focused around the experience of the players. During testing, we encouraged the players to be vocal about their intentions as they played, to better understand both what the players *thought* they could as well as what they *wanted* to do. Focusing first around managing expectations and granting affordances the players believed they should have helped create a positive experience for players playing *Xeero.*

Communication between Eric and Dan was vital for developing art assets that contributed to the cohesive whole of the game. By discussing new features, mechanics, and objects, and the motivations behind them, the two were able to collaborate in the design of the audio experience of *Xeero*.

Working in a small group also fostered a cohesive creative vision for the game. Eric, with hands in both code and art, was able to create a game that displays a unified vision in character, prop, and environment design. The foundational mechanics of the game -- platforming, digitizing, and combat -- are iterated and combined over the course of the game to create new mechanics in a way that relates to the overall experience of the game.

8.1.2. Dan Acito

Overall the group worked together really well. We were always good about communicating with each other and making sure everyone knew where everyone else was in terms of work. Whenever Dan would finish up some of the sound effects, he would push them into the repository so that Eric could make sure that the sound cues were hooked into the code. The same applied the other way around as well.

Whenever Eric added a new obstacle or mechanic to the game, he would demo it to Dan, and would describe the general functionality and intention of his new addition. This would help Dan get a sense of what kind of sound effect he should make, and how to time it appropriately. These meetings helped keep the two in sync throughout the project, which ultimately increased productivity.

8.1.3. Anthony Sessa

Despite joining the project later in development, getting up to speed on what was already accomplished as well as what still had to be done was simple. The repository had all of the pertinent information readily available, and anything that had to be clarified was done so by Eric or Dan. The team worked very well together. Meetings were loose and laid back, and figuring out logistics for the various showcases for *Xeero* was effortless. Having previous experience producing a student project really helped Anthony in terms of how to go about creating timelines and moving the project along. Most importantly, he got more experience working in a production role and several pieces of production material he could use in the future in a portfolio for potential employers.

8.2. What Went Wrong

8.2.1. Eric Anumba

Overall, the project suffered from many of the issues that student projects tend to face: improper planning and issues defining scope. Eric designed the features he wanted to implement within the game, but it was quickly apparent that the scope of the game was larger than what the team could reasonably complete. Eric and Dan created an approximate monthly schedule early in their collaboration, but quickly the project missed milestones, and several times over the course of the project, the game had to be re-scoped to be completed within the time frame of the Master's Project.

Additionally, the plans the team created did not account for the time required to prepare festivals and exhibitions. Many times over the project, development plans were foregone to prepare for the exhibitions at which Xeero was shown.

The initial choice of framework for the game and its engine, Microsoft XNA, played to the disadvantage of the project. By the start of the project's creation, Microsoft had announced that they would discontinue the framework, and the framework was officially dropped while the Master's Project was in-progress. Eric had researched replacement technology, focusing on MonoGame to allow the game to reach multiple platforms, but even the two frameworks, sharing a nearly identical, will incur a high time cost to switch between them. Through Eric's preliminary tests, the framework MonoGame misses some features vital to the performance of the game (like hardware instancing), while other large, complex parts of engine (like the shaders used to create the game's aesthetic) would need to be rewritten.

8.2.2. Dan Acito

The biggest mistake that Dan made throughout the development process was not being fully organized from the beginning. By not following a strict naming convention with his .wav files, and not categorizing the sound bank, it became difficult to keep track of information. While this issue was remedied before too much progress had been made, it still ended up costing Dan valuable time by having to go through and rename all of the sound files.

For the music, Dan was severely limited by his hardware choices. Even though the *Rock Band* keyboard functioned well at what it was needed to do, it fell far behind in terms of what a professional quality MIDI controller could do. Due to the fact that it wasn't explicitly designed for music production, the interface was a little confusing. Certain buttons on the controller could control many different things such as octave or velocity, but it wasn't clear which button did what unless Dan guessed. Since the controller is designed specifically for gameplay, the buttons weren't labelled for production, resulting in Dan doing a lot of guess work. While Dan was still successful in utilizing these tools, the extra effort needed to figure out the interface was slightly inefficient.

8.2.3. Anthony Sessa

As Eric mentioned, planning for the festivals that *Xeero* was showcased at often took priority over development. The team was so focused on getting the game in front of the public that priorities often got shifted. In addition, meetings often had to be scheduled for later at night, as Anthony had a typical 9-5 full time job that took up a large amount of time. Meetings occasionally had to be conducted via Skype which didn't offer the same advantages that in person meetings would have. Anthony also ran into trouble getting the game entered into the PAX booth at first since the booth runners were reluctant to include *Xeero* two years in a row.

8.3. What we would do differently

8.3.1. Eric Anumba

From the onset, the design and production of the game should follow a more rigid methodology. Most of the design of *Xeero* occurred after much of the core functionality and visual style was implemented, restricting the potential gameplay and story options, and forcing the design of the game around the "computer" motif. Initially, the game's experience goals and target audience should be more clearly defined, and the game should be more thoroughly designed and more lightly prototyped to allow for experimentation in finding mechanics that would fulfill the aforementioned goals and meet the needs of the audience.

Over the development of the game, several features were implemented that eventually did not make an appearance in the first installment of the game (like other digitizable objects, bombs and unstable blocks). In later projects, the production of the game should follow a formal methodology to define features to implement and milestones to reach to help minimize unnecessary development and prioritize important functionality.

8.3.2. Dan Acito

If within the budget, Dan would have invested in some better hardware to utilize for the music production. While the cables and audio interface that Dan borrowed were in pristine condition, they did not do justice to the low quality keyboard controller that Dan was using. Given Dan's limitations, the music quality turned out quite good, but it easily could have been much better with higher quality tools.

Now that Dan has had some experience using asset lists for sound design, he knows that organization is extremely important. In future projects, not only will he follow recognizable naming conventions, but he also will separate sounds into different categories so that they are easier to keep track of.

8.3.3. Anthony Sessa

Anthony would have liked to have gotten into contact with Eric and Dan a little earlier in the summer than he did. This would have allowed him to get up to speed a little earlier and make the first couple of weeks planning out the first semester a little easier. In addition, while OpenProj (a free to use application similar to Microsoft Project) was suitable for planning out production schedules, Anthony would have liked to have gotten his hands on a license for Microsoft Project since the far majority of companies in the industry use it in one form or another and being able to show that on his resume would have been a huge plus. Finally, Anthony wanted to budget more time for the team to prepare a Xeero presentation for the MassDigi Game Challenge in Boston. The team was so focused on furthering the development of the game and adding content that the deadline for the contest snuck up. The game was just a couple of extra days away from being ready to show at the contest and being able to get in there would have been a nice accolade for the team.

9. Conclusion

For this project, three team members, Eric Anumba, Daniel Acito, and Anthony Sessa, worked to refine the game *Xeero* as their IMGD Master's Project. The game, an action-puzzle-platformer developed for the PC, will eventually be commercially released in episodic installments, and has been exhibited at PAX East and other local Massachusetts conventions. The game is designed for fans of 3D action and platforming games, and seeks to evoke a sense of satisfaction of the player through two essential moments of gameplay: acquiring new, exciting abilities and the experiencing the interplay of different of game mechanics to deftly accomplish a goal.

In developing *Xeero*, a game engine was developed that implements functionality for required the core of the game: rendering, physics, audio, content management, particle systems, and GUI creation, and also contains a level editor. Additionally, an art style, design strategy, and audio creation methodology was created for *Xeero*. These foundations can be used to rapidly develop new content for the game.

The game is broken into six planned parts, and this Master's Project is designed to build and refine the first episode. For the future, the remaining installments need to be developed, tested, and released.

Works Cited

Acev. "Review: Tales of Symphonia (GCN)." *Business in Mayhem*. 30 July 2012. Web. 27 Apr. 2015. <http://dowase.net/blog/?p=374>.

Bastion. Warner Bros. Interactive Entertainment. 20 Jul. 2011. Video game.

Bayonetta. Nintendo. 5 Jan. 2010. Video game.

Birkhead, Mike. "Depth vs Breadth in Combat Design: An Interactive Visualization." *Flark Design*. 25 May 2011. Web. <http://www.flarkminator.com/2011/05/25/depth-vs-breadth-in-combat-design-an-interactive-visualization/>.

Birkhead, Mike. "Opinion: What Makes Combat Fun." *Opinion: What Makes Combat Fun*. Gamasutra, 21 Sept. 2011. Web. <http://www.gamasutra.com/view/news/37356/Opinion_What_Makes_Combat_Fun.php>.

C., Radford. "Bastion Review (Xbox, PC) - Page 2 of 2 - Lazy Tech Guys." *Lazy Tech Guys*. 22 Aug. 2011. Web.<http://lazytechguys.com/reviews/bastion-review-xbox-pc/2>.

Chambers. "Dust: An Elysian Tail on PS4 Is... Adequate." *Dust: An Elysian Tail*. Game Skinny, 29 Oct. 2014. Web. 27 Apr. 2015. <http://www.gameskinny.com/7qz3d/dust-an-elysian-tail-on-ps4-is-adequate>.

Darksiders. Nordic Games. 5 Jan. 2010. Video game.

Dust: An Elysian Tail. Microsoft Studios. 15 Aug. 2012. Video game.

Francis, Geoffrey. Cockos. *Reaper User Guide v. 4.76.* Dec 2014. Web. <http://dl.reaper.fm/userguide/ReaperUserGuide476C.pdf>

"Getting Started with XACT." *Getting Started with XACT*. Microsoft Corporation. Web. <https://msdn.microsoft.com/en-us/library/ff827592.aspx>.

God of War. Sony Computer Entertainment. 22 Mar. 2005. Video game.

Hruska, Joel. "Microsoft Kills Xbox 360/PC Cross-platform Development, Declares DirectX “no Longer Evolving” | ExtremeTech." *ExtremeTech*. 1 Feb. 2013. Web. <http://www.extremetech.com/gaming/147289-microsoft-kills-xbox-360pc-cross-platform-development-declares-directx-no-longer-evolving>.

Jackson, Simon. "XNA Is No More, as the Phoenix Rises from the Ashes." *Dark Genesis*. 17 Mar. 2015. Web. 27 Apr. 2015. <http://darkgenesis.zenithmoon.com/xna-is-no-more-as-the-phoenix-rises-from-the-ashes/>.

JoouChiyanmon. *Cyberspace Move*. Digital image. *DeviantArt*. Web.

JusticeVonBrandt. *Cyberspace*. Digital image. *DeviantArt*. Web. Levin, Raph. “Inconsolata.” Web.

Kingdom Hearts II. Square Enix. 22 Dec. 2005. Video game.

Ninjatic. *Floating Ruins*. Digital image. *DeviantArt*. Web.

Northernlion. “Let's Play - Bastion - Episode 1 [The Rippling Walls]” Online video clip. *YouTube*. Google, 9 Dec. 2011. Web.

Nubik. "Spritefont 2 Texture Tool." Web.

Portal. Valve Corporation. 9 Oct. 2007. Video game.

"Portal 2 Puzzle Maker: Reflection Cube." *Valve Developer Community*. Valve Corporation. Web. <https://developer.valvesoftware.com/wiki/Portal_2_Puzzle_Maker/Reflection_Cube>.

"Real-Time Physics Simulation." *Bullet Physics Library*. RealTime Physics Simulation. Web. <http://bulletphysics.org/>.

RENMIRI. "Final Fantasy X - Spira's Tales." *Final Fantasy X - Spira's Tales*. 20 Jan. 2006. Web. 27 Apr. 2015. <http://spirablog.blogspot.com/2006/01/quidditch-versus-blitzball.html>.

Stolpe, Philip. "About." *About*. MonoGame. Web. <http://www.monogame.net/about/>.

Tales of Symphonia. Namco. 29 Aug. 2003. Video game. Thomas, Frank, and Ollie Johnston. *The Illusion of Life: Disney Animation*. New York: Disney Editions, 1995. Print.

Traks, Andres. "BulletSharp." *BulletSharp*. Web. 27 Apr. 2015. <http://andrestraks.github.io/BulletSharp/>.

Trine. Nobilis. 3 Jul. 2009. Video game.

TypoCity. *Cyberspace*. Digital image. *DeviantArt*. Web.

Typodermic Fonts. “Quadrangle.” Web.

venomblade891. “PCSX2- Kingdom Hearts II Final Mix (English Patched) Lingering Sentiment Battle [Terra's Armor].” Online video clip. *YouTube*. Google, 11 Sep. 2011. Web.

WastedMeerkat. “Kingdom Hearts II - Final Form Training Route” Online video clip. *YouTube*. Google, 25 Sep. 2011. Web.

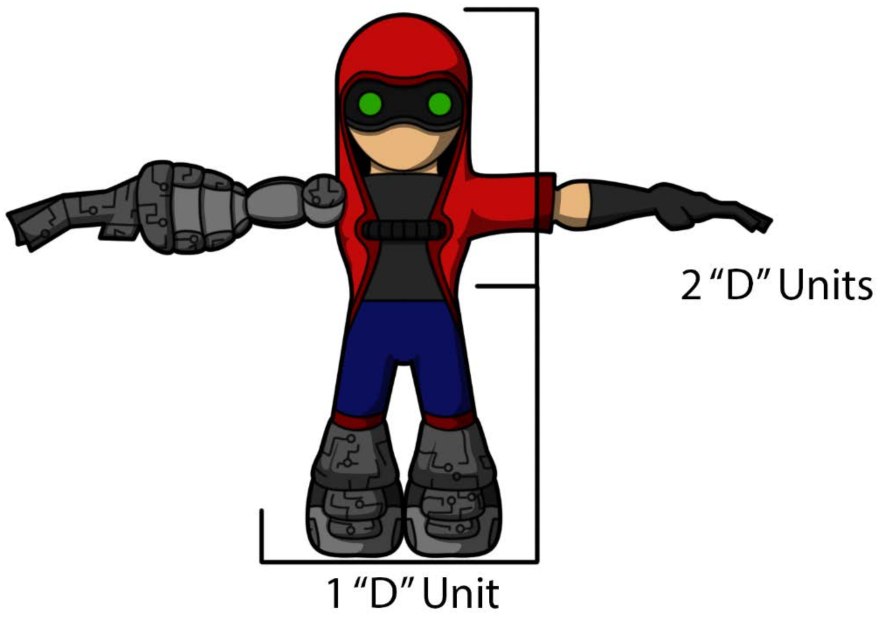
Yablonski, Kirby. "Darksiders II (Wii U) Review." *Canadian Online Gamers Darksiders II Wii U Review Comments*. Canadian Online Gamers Network. Web. 27 Apr. 2015. <http://canadianonlinegamers.com/review/darksiders-ii-wii-u-review/>.

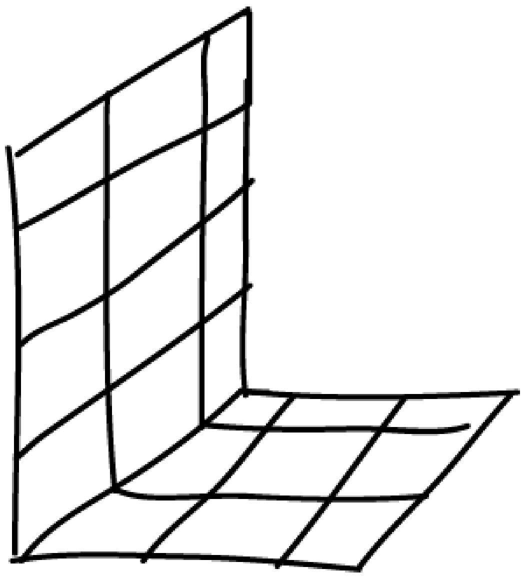
Appendix A: Initial Game Metrics

Debugger is a platformer and is focused around moving the character, D, around the world, so every object in the game must be built around D's size.

D Unit

Every object in the game is sized with the "D Unit," a size directly related to D's proportions. The character is about 1 unit wide and deep, and 2 units tall.





Maps

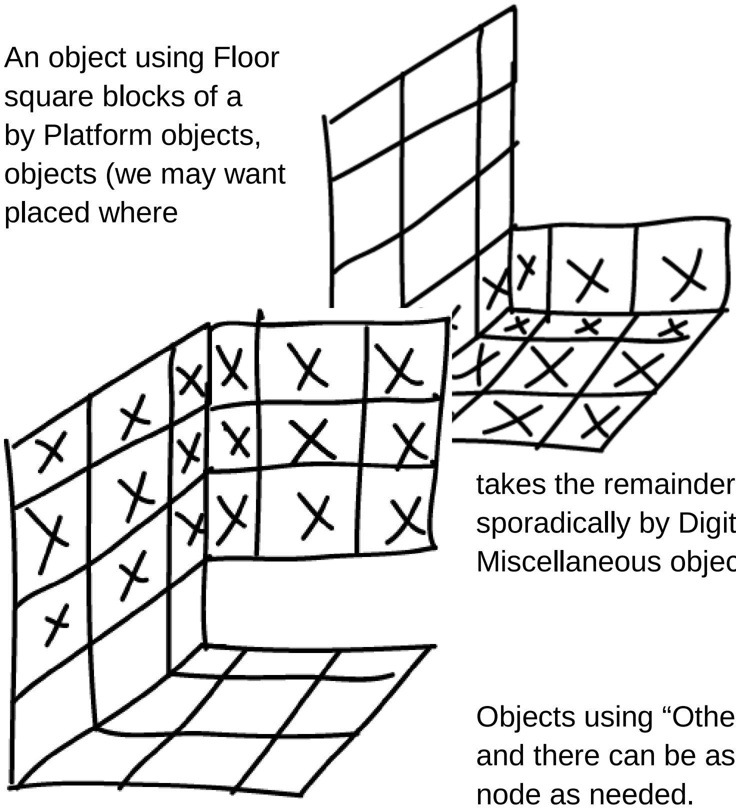
Maps in this game are broken up into nodes (LevelNode) that contain the various objects that D can interact with.
Each LevelNode is 3x3x4 D units (four units tall), and each node is flush with each of its adjacent nodes.

Node Positioning

For map building, there are specific areas of the node assigned to specific types of objects. This area is represented by the NodePositioning enumeration that is present in all WorldObjects.

NodePositioning can either take on “Floor,” “Center,” or “Other.”

An object using Floor Positioning takes up the bottom nine square blocks of a node. Floor Positioning is used mostly by Platform objects, but isn’t restricted to just those objects (we may want to include an obstacle object that is placed where platforms usually go).



An object using Center Positioning takes the remainder of the node; Center Positioning is used sporadically by DigitizableObjects, Obstacles, and Miscellaneous objects.

Objects using “Other” don’t take up any space on the node, and there can be as many of these types of objects on a node as needed.

Other Positioning is used mostly by objects that don’t have anything physical appear in the world, like checkpoints, finish points, or DO Spawn Points, or objects that don’t use any kind of collision detection, like scenery objects.

Objects using either of these positioning schemes do not need to take up *all* of their allotted space, and this information is not used for collision detection. This is solely used for map building, so the level parser knows the limit of objects in a node, and can throw errors as needed.

Movement Metrics

The player is assumed to be able to jump up at least as high as the top of the Block DO (so it can be used to move up a level), and can clear three empty nodes space. The player can also double jump, so the player must be able to jump 1.75 units up and 5.25 units across.

The two things that affect these distances are jump velocity (which relies entirely on gravity) and run speed. If we fix a desired run speed (like 2-3 units/second), then the world’s gravity becomes a function of the run speed, jump height, and jump distance.

Map Coordinates

When maps are being built, the parser reads each object in Node coordinates, where each point (x,y,z) refers to a node to address in a 3D grid. X, Y, Z are integers, and a node at (0,0,0) is directly adjacent to (1,0,0).

Up is defined as <0,1,0>, and the game uses the right-hand coordinate system.

Appendix B: *Xeero* Promotional Poster

XEERO™
void Fuse(bool Fuse){Vertices
;for(i = 0; i < v.Count; i++)

Appendix C: Sound List

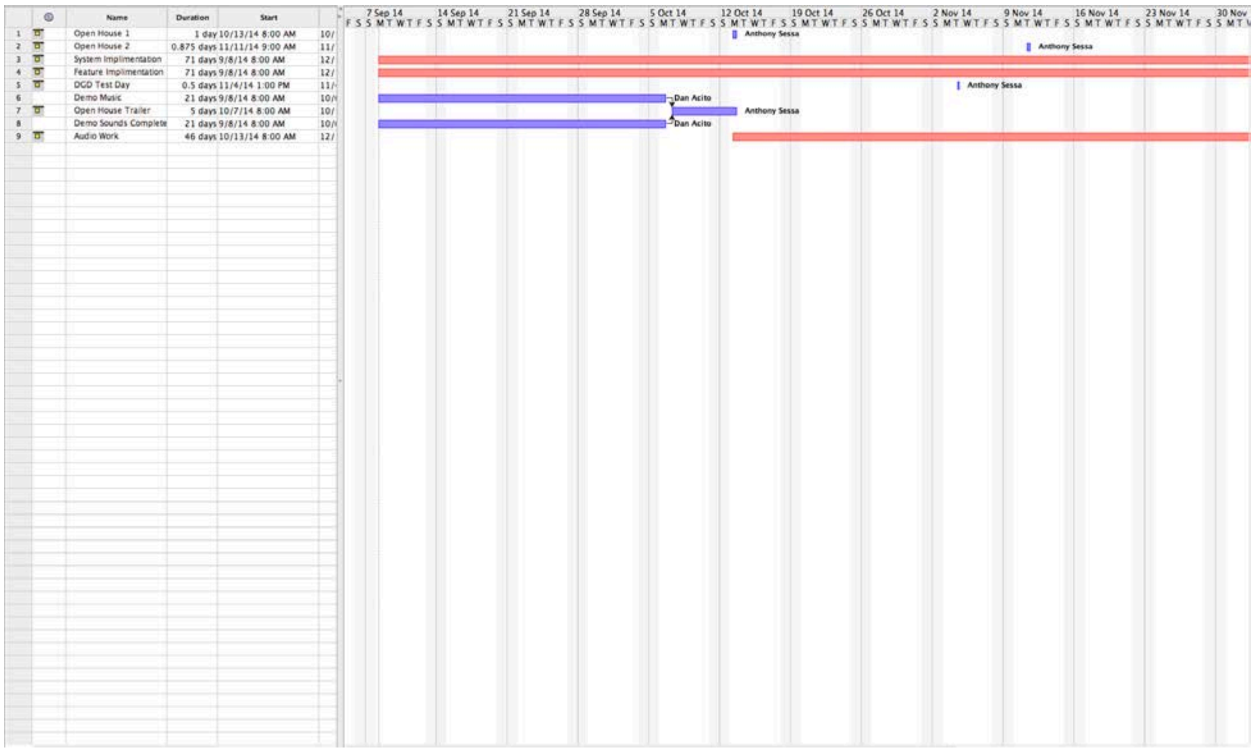
Sound	CueName	Music/Effect	Functionality In Game?	Loop?	Discussed	Bank	File-Created	Edited	In Sound Engine	Playtested
Music										
Title Music	TitleMusic	M	N	Y	N	Music	N	N	N	N
World 1 Combat Music	Combat1	M	Y	Y	Y	Music	Y	Y	Y	N
World 1 Level Music	Level1	M	Y	Y	Y	Music	Y	Y	Y	N
World 1 Hub Music	Hub1	M	Y	Y	Y	Music	Y	Y	Y	N
World 1 Boss Music	Boss1	M	N	Y	Y	Music	Y	Y	N	N
Near Bug Music?	NearBug1	M	N	Y	N	Music	N	N	N	N
Tech Circles										
Digitize Circle	CircleDigitize	E	Y	N	Y	Levels	Y	Y	Y	N
Fuse Circle	CircleFuse	E	Y	N	Y	Levels	Y	Y	Y	N
Create Circle	CircleCreate	E	Y	N	Y	Levels	Y	Y	Y	N
Switch Circle (weapons)	CircleSwitch	E	Y	N	Y	Levels	Y	Y	Y	N
Upgrade Circle	CircleUpgrade	E	Y	N	Y	Levels	Y	Y	Y	N
Digitize Hexagons	DigitizeHexagon	E	Y	N	Y	Levels	Y	Y	Y	N
Despawn "Pop"	DespawnPop	E	Y	N	Y	Levels	Y	Y	Y	N
Player										
Digitize Player	PlayerDigitize	E	Y	Y	Y	Levels	Y	Y	Y	N
Materialize Player	PlayerMaterialize	E	Y	Y	Y	Levels	Y	Y	Y	N
Walking Footstep	PlayerFootstepWalk	E	Y	N	Y	Levels	Y	N	Y	N
Landing Footstep	PlayerFootstepLand	E	Y	N	Y	Levels	Y	N	Y	N
High Landing Footstep	PlayerFootstepHighLand	E	Y	N	Y	Levels	Y	N	Y	N
Ledge Grab Hand Impact	PlayerHandImpactLedge	E	Y	N	Y	Levels	Y	Y	Y	N
Jump	PlayerJump	E	Y	N	Y	Levels	Y	Y	Y	N
Double Jump	PlayerDoubleJump	E	Y	N	Y	Levels	Y	Y	Y	N
Wall Jump	PlayerWallJump	E	Y	N	Y	Levels	Y	Y	Y	N
Ledge Jump	PlayerLedgeJump	E	Y	N	Y	Levels	Y	Y	Y	N
Block Start	PlayerBlockStart	E	Y	N	Y	Levels	Y	Y	Y	N
Blocking (Weapon Spin)	PlayerBlockSpin1H	E	Y	Y	Y	Levels	Y	Y	Y	N
Dual Wield Blocking (Weapon Spin) - Pri	PlayerBlockSpin2HPrimary	E	Y	Y	Y	Levels	N	N	N	N
Dual Wield Blocking (Weapon Spin) - Sec	PlayerBlockSpin2HSecondary	E	Y	Y	Y	Levels	N	N	N	N
Dodge Roll (ground)	PlayerDodgeRoll	E	Y	N	Y	Levels	Y	N	Y	N
Dodge Spin (mid-air)	PlayerDodgeSpin	E	Y	N	Y	Levels	Y	Y	Y	N
Player Burn Damage	PlayerDamageBurn	E	Y	N	Y	Levels	N	N	N	N
Player Hit Damage	PlayerDamageHit	E	Y	N	Y	Levels	N	N	N	N
Player Knockback	PlayerDamageKnockback	E	Y	N	Y	Levels	N	N	N	N
Player Glitch Sound	PlayerGlitch	E	Y	N	Y	Levels	Y	Y	Y	N
Idle Animation (Look)	PlayerIdleLook	E	Y	N	Y	Levels	Y	Y	Y	N
Idle Animation (Data) - Clothes Start	PlayerIdleDataClothesStart	E	N	N	Y	Levels	Y	Y	Y	N
Idle Animation (Data) - Clothes End	PlayerIdleDataClothesEnd	E	N	N	Y					
Idle Animation (Data) - Glow	PlayerIdleDataGlow	E	N	N	Y					
Idle Animation (Data) - Blocks	PlayerIdleDataBlocks	E	N	N	Y					
Weapon Switch	PlayerWeaponSwitch	E	Y	Y	N	Levels	N	N	N	N
Weapon Switch Spin	PlayerWeaponSwitchSpin	E	Y	N	N	Levels	N	N	N	N
Wall Slide	PlayerWallSlide	E	Y	Y	N	Levels	Y	N	Y	N
Dying	PlayerDying	E	Y	N	Y	Levels	Y	Y	Y	N
Bug Crush	PlayerBugCrush	E	Y	N	Y	Levels	Y	Y	Y	N
Launch	PlayerLaunch	E	Y	N	N	Levels	Y	Y	Y	N
Attacks										
Attack Wait Spin	PlayerAttackWaitSpin	E	Y	Y	N	Levels	Y	Y	Y	N
Reaper										
Reaper 1 - 1H Ground - Primary	PlayerReaper1_1G_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper 2 - 1H Ground - Primary	PlayerReaper2_1G_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper Finisher - 1H Ground - Primary	PlayerReaperFinisher_1G_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper 1 - 1H Air - Primary	PlayerReaper1_1A_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper 2 - 1H Air - Primary	PlayerReaper2_1A_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper Finisher - 1H Air - Primary	PlayerReaperFinisher_1A_Primary	E	Y	N	Y	Levels	Y	N	N	N
Reaper 1 - 2H Ground - Primary	PlayerReaper1_2G_Primary	E	N	N	N	Levels	N	N	N	N
Reaper 1 - 2H Ground - Secondary	PlayerReaper1_2G_Secondary	E	N	N	N	Levels	N	N	N	N
Reaper 2 - 2H Ground - Primary	PlayerReaper2_2G_Primary	E	N	N	N	Levels	N	N	N	N
Reaper 2 - 2H Ground - Secondary	PlayerReaper2_2G_Secondary	E	N	N	N	Levels	N	N	N	N
Reaper Finisher - 2H Ground - Primary	PlayerReaperFinisher_2G_Primary	E	N	N	N	Levels	N	N	N	N
Reaper Finisher - 2H Ground - Secondary	PlayerReaperFinisher_2G_Secondary	E	N	N	N	Levels	N	N	N	N
Reaper 1 - 2H Air - Primary	PlayerReaper1_2A_Primary	E	N	N	N	Levels	N	N	N	N
Reaper 1 - 2H Air - Secondary	PlayerReaper1_2A_Secondary	E	N	N	N	Levels	N	N	N	N
Reaper 2 - 2H Air - Primary	PlayerReaper2_2A_Primary	E	N	N	N	Levels	N	N	N	N
Reaper 2 - 2H Air - Secondary	PlayerReaper2_2A_Secondary	E	N	N	N	Levels	N	N	N	N
Reaper Finisher - 2H Air - Primary	PlayerReaperFinisher_2A_Primary	E	N	N	N	Levels	N	N	N	N
Reaper Finisher - 2H Air - Secondary	PlayerReaperFinisher_2A_Secondary	E	N	N	N	Levels	N	N	N	N
Corrupted Object										
Glitch	CorruptedObjectGlitch	E	Y	N	Y	Levels	Y	Y	Y	N
Weapons										
Digitize Weapon	WeaponDigitize	E	Y	Y	Y	Levels	Y	Y	Y	N
Materialize Weapon	WeaponMaterialize	E	Y	Y	Y	Levels	Y	Y	Y	N
Weapon Stick	WeaponStick	E	Y	N	N	Levels	Y	Y	Y	N
Weapon Break	WeaponBreak	E	Y	N	Y	Levels	N	N	N	N
Weapon Bounce	WeaponBounce	E	Y	N	Y	Levels	N	N	N	N
Upgrade Chip										
Digitize Upgrade Chip	UpgradeChipDigitize	E	Y	Y	Y	Levels	Y	Y	Y	N
Materialize Upgrade Chip	UpgradeChipMaterialize	E	Y	Y	Y	Levels	Y	Y	Y	N
Upgrade Chip Idle	UpgradeChipIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
Upgrade Chip Bounce	UpgradeChipBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Effect Initial Burst	UpgradeEffectInitialBurst	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Effect Beam Burst	UpgradeEffectBeamBurst	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Effect Finale Burst	UpgradeEffectFinaleBurst	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Light Beam Start	UpgradeEffectLightBeamStart	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Light Beam Middle	UpgradeEffectLightBeam	E	Y	Y	Y	Levels	Y	Y	Y	N
Acquired Upgrade Light Beam End	UpgradeEffectLightBeamEnd	E	Y	N	Y	Levels	Y	Y	Y	N
Acquired Upgrade Effect Digit Steam	UpgradeEffectDigitSteam	E	Y	Y	Y	Levels	Y	Y	Y	N
Info Pages										
Info Page Cutscene Image	InfoPageCutsceneImage	E	Y	N	Y	Levels	Y	Y	Y	N
Info Page Cutscene Description	InfoPageCutsceneDesc	E	Y	N	Y	Levels	Y	Y	Y	N
Control Prompt										
Control Prompt Pop	ControlPromptPop	E	Y	N	Y	Levels	Y	Y	Y	N
Control Prompt TechLines	ControlPromptTechLines	E	Y	N	Y	Levels	Y	Y	Y	N
Combat										
Block Impact	ImpactBlock	E	Y	N	Y	Levels	Y	Y	Y	N
Block Broken Impact	ImpactBlockBroken	E	Y	N	Y	Levels	Y	Y	Y	N
Counter Impact	ImpactCounter	E	Y	N	Y	Levels	Y	Y	Y	N
Evade Impact	ImpactEvade	E	Y	N	Y	Levels	Y	Y	Y	N

Digitizable Objects										
Digitizable Object Disassembly	DODisassembly	E	Y	N	Y	Levels	Y	Y	Y	N
Digitizable Object Impact	DOImpact	E	Y	N	Y	Levels	Y	Y	Y	N
Digitizable Object Purge Indicator	DOPurgeIndicator	E	Y	N	Y	Levels	Y	N	Y	N
Digitizable Object Purge	DOPurge	E	Y	Y	Y	Levels	Y	Y	Y	N
Blocks										
Digitize Block	BlockDigitize	E	Y	Y	Y	Levels	Y	Y	Y	N
Materialize Block	BlockMaterialize	E	Y	Y	Y	Levels	Y	Y	Y	N
Bombs										
Digitize Bomb	BombDigitize	E	Y	Y	Y	Levels	Y	Y	Y	N
Materialize Bomb	BombMaterialize	E	Y	Y	Y	Levels	Y	Y	Y	N
Bomb Pre-Detonation	BombPreDetonate	E	Y	Y	Y	Levels	Y	Y	Y	N
Bomb Bounce	BombBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Bomb Detonation	BombDetonate	E	Y	N	Y	Levels	Y	N	Y	N
Rockets										
Digitize Rocket	RocketDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Rocket	RocketMaterialize	E	N	Y	Y	Levels	N	N	N	N
Magnets										
Digitize Magnet	MagnetDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Magnet	MagnetMaterialize	E	N	Y	Y	Levels	N	N	N	N
Gravity Field										
Digitize Gravity Field	GravityFieldDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Gravity Field	GravityFieldMaterialize	E	N	Y	Y	Levels	N	N	N	N
Unstable Block										
Digitize Unstable Block	UBlockDigitize	E	Y	Y	Y	Levels	Y	N	N	N
Materialize Unstable Block	UBlockMaterialize	E	Y	Y	Y	Levels	Y	N	N	N
Unstable Block Idle	UBlockIdle	E	Y	N	Y	Levels	Y	N	N	N
Unstable Block Detonation	UBlockDetonate	E	Y	N	Y	Levels	Y	N	N	N
Moving Block										
Digitize Moving Block	MovingBlockDigitize	E	Y	Y	Y	Levels	N	N	N	N
Materialize Moving Block	MovingBlockMaterialize	E	Y	Y	Y	Levels	N	N	N	N
Magnet Platform										
Digitize Magnet Platform	MagnetPlatformDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Magnet Platform	MagnetPlatformMaterialize	E	N	Y	Y	Levels	N	N	N	N
Gravity Platform										
Digitize Gravity Platform	GravityPlatformDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Gravity Platform	GravityPlatformMaterialize	E	N	Y	Y	Levels	N	N	N	N
Missile										
Digitize Missile	MissileDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Missile	MissileMaterialize	E	N	Y	Y	Levels	N	N	N	N
Heat-Seeking Bomb										
Digitize "Heat-Seaking" Bomb	HeatBombDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize "Heat-Seaking" Bomb	HeatBombMaterialize	E	N	Y	Y	Levels	N	N	N	N
Gravity Bomb										
Digitize Gravity Bomb	GravityBombDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Gravity Bomb	GravityBombMaterialize	E	N	Y	Y	Levels	N	N	N	N
Moving Magnet										
Digitize Moving Magnet	MovingMagnetDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Moving Magnet	MovingMagnetMaterialize	E	N	Y	Y	Levels	N	N	N	N
Moving Gravity Field										
Digitize Moving Gravity Field	MovingGravityFieldDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Moving Gravity Field	MovingGravityFieldMaterialize	E	N	Y	Y	Levels	N	N	N	N
Gravity Pulse										
Digitize Gravity Pulse	GravityPulseDigitize	E	N	Y	Y	Levels	N	N	N	N
Materialize Gravity Pulse	GravityPulseMaterialize	E	N	Y	Y	Levels	N	N	N	N
Attackable Blockades										
Attackable Blockade Hit	ABlockadeHit	E	Y	N	N	Levels	Y	Y	Y	N
Attackable Blockade Destruction	ABlockadeDestroy	E	Y	N	N	Levels	Y	Y	Y	N
Destructible Blockades										
Destructible Blockade Destruction	DBlockadeDestroy	E	Y	N	Y	Levels	Y	N	N	N
Archive										
Archive Idle (Hum)	ArchiveIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
Archive Destruction	ArchiveDestroy	E	Y	N	Y	Levels	Y	Y	Y	N
Archive Bounce	ArchiveBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Big Archive										
Big Archive Idle (Hum)	BigArchiveIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
Big Archive Hit	BigArchiveHit	E	Y	N	Y	Levels	Y	Y	Y	N
Big Archive Destruction	BigArchiveDestroy	E	Y	N	Y	Levels	Y	Y	Y	N
Big Archive Bounce	BigArchiveBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Falling Objects (Platform/Side Platform)										
Falling Object Return to Idle	FallingObjectReturn	E	Y	N	Y	Levels	Y	Y	Y	N
Falling Object Vibrating	FallingObjectVibrate	E	Y	N	Y	Levels	Y	Y	Y	N
Falling Object Falling	FallingObjectFalling	E	Y	Y	Y	Levels	Y	Y	Y	N
Bug										
Bug Idle Sound	BugIdle	E	Y	N	Y	Levels	Y	Y	Y	N
Debris Object										
Debris Clatter/Collision	DebrisBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Health Pixel										
Health Pixel Clatter/Collision	HealthPixelBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Health Gain	HealthGain	E	Y	N	Y	Levels	Y	Y	Y	N
Health Pickup	HealthPickup[n]	E	Y	N	Y	Levels	Y	Y	Y	N
BitBucks										
BitBucks Clatter/Collision	BitBucksBounce	E	Y	N	Y	Levels	Y	Y	Y	N
BitBucks Pickup	BitBucksPickup	E	Y	N	Y	Levels	Y	Y	Y	N
Enemies										
Enemy Spawn	EnemySpawn	E	Y	Y	Y	Levels	Y	Y	Y	N
Enemy Despawn	EnemyDespawn	E	Y	Y	Y	Levels	Y	Y	Y	N
Enemy Dodging	EnemyDodging	E	Y	N	Y	Levels	Y	Y	Y	N
Viruses										
Virus Replicate Glow	VirusReplicateGlow	E	Y	Y	Y	Levels	Y	Y	Y	N
Virus Replicate "Pop"	VirusReplicatePop	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus										
Spider Virus Lens Movement	SpiderVirusLens	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Swipe	SpiderVirusSwipe	E	Y	N	Y	Levels	Y	Y	Y	N

Spider Virus Head Spin	SpiderVirusHeadSpin	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Hit Sound	SpiderVirusDamageHit	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Knockback	SpiderVirusDamageKnockback	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Burn Damage	SpiderVirusDamageBurn	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Landing	SpiderVirusLanding	E	Y	N	Y	Levels	Y	Y	Y	N
Spider Virus Leaping	SpiderVirusLeaping	E	Y	N	Y	Levels	Y	Y	Y	N
Soldier Virus										
Soldier Virus Hit Sound	SoldierVirusDamageHit	E	Y	N	Y	Levels	Y	Y	Y	N
Soldier Virus Knockback	SoldierVirusDamageKnockback	E	Y	N	Y	Levels	Y	Y	Y	N
Soldier Virus Burn Damage	SoldierVirusDamageBurn	E	Y	N	Y	Levels	Y	Y	Y	N
Soldier Virus Landing	SoldierVirusLanding	E	Y	N	Y	Levels	Y	Y	Y	N
Soldier Virus Leaping	SoldierVirusLeaping	E	Y	N	Y	Levels	Y	Y	Y	N
Laser Turret										
Laser Turret Clatter/Collision	LaserTurretBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Hit Switch										
Hit Switch Button	HitSwitchButton	E	Y	N	Y	Levels	Y	Y	Y	N
Button										
Button Trigger	ButtonTrigger	E	Y	N	Y	Levels	Y	Y	Y	N
Block Switch										
Block Switch Trigger	BlockSwitchTrigger	E	Y	N	Y	Levels	Y	Y	Y	N
Timed Switches										
Timed Switch Tick	TimedHitSwitchTick	E	Y	N	Y	Levels	Y	Y	Y	N
Timed Switch Failed	TimedSwitchFailed	E	Y	N	Y	Levels	Y	Y	Y	N
Tethered Hit Switch										
Tethered Hit Switch Hit	TetheredHitSwitchHit	E	N	N	Y	Levels	N	N	N	N
Gates										
Gate Door Bounce	GateDoorBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Bit Slicers										
Bit Slicer Bounce	BitSlicerBounce	E	Y	N	Y	Levels	Y	Y	Y	N
Bit Slicer Slice Down	BitSlicerSliceDown	E	Y	Y	Y	Levels	Y	Y	Y	N
Bit Slicer Slice Up	BitSlicerSliceUp	E	Y	Y	Y	Levels	Y	Y	Y	N
Crusher										
CrusherIdle	CrusherIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
CrusherIdleStart (2 seconds)	CrusherIdleStart	E	Y	N	Y	Levels	Y	Y	Y	N
CrusherIdleStop (2 seconds)	CrusherIdleStop	E	Y	N	Y	Levels	Y	Y	Y	N
CrusherImpact	CrusherImpact	E	Y	N	Y	Levels	Y	Y	Y	N
CrusherDOImpact	CrusherDOImpact	E	Y	N	Y	Levels	Y	Y	Y	N
Purge Gate										
Purge Gate Idle (Discuss proximity hum)	PurgeGateIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
Vending Machine										
Vending Machine Working	VendingMachineWorking	E	Y	Y	Y	Levels	Y	Y	Y	N
Vending Machine Bump	VendingMachineBump	E	Y	N	Y	Levels	Y	Y	Y	N
Vending Machine Dispense	VendingMachineDispense	E	Y	N	Y	Levels	Y	Y	Y	N
Boss Portal										
Boss Portal Engage	BossPortalEngage	E	Y	N	Y	Levels	Y	Y	Y	N
Boss Portal Idle	BossPortalIdle	E	Y	Y	Y	Levels	Y	Y	Y	N
Boss Portal Enter	BossPortalEnter	E	Y	N	Y	Levels	Y	Y	Y	N
GUI/Menus										
Menu Button Mouse Hover	MenuButtonMouseHover	E	Y	N	Y	Levels	Y	Y	Y	N
Menu Button Key Select	MenuButtonKeySelect	E	Y	N	Y	Levels	Y	Y	Y	N
Menu Button Press	MenuButtonPress	E	Y	N	Y	Levels	Y	Y	Y	N
Menu Button Trigger	MenuButtonTrigger	E	Y	N	Y	Levels	Y	Y	Y	N
HUD										
Health Bar Grow	HUDHealthBarGlow	E	Y	N	Y	Levels	Y	Y	Y	N
DO Slot Fade Up	HUDDOSlotFadeUp	E	Y	N	Y	Levels	Y	Y	Y	N
DO Slot Rings Fade Up	HUDDOSlotRingsFadeUp	E	Y	N	Y	Levels	Y	Y	Y	N
DO Slot Rings Expand	HUDDOSlotRingsExpand	E	Y	N	Y	Levels	Y	Y	Y	N
DO Slot Rings Contract	HUDDOSlotRingsContract	E	Y	N	Y	Levels	Y	Y	Y	N
DO Slot Rings Glow	HUDDOSlotRingsGlow	E	Y	N	Y	Levels	Y	Y	Y	N
Fused DO Slot Energy Charge	HUDFusedDOSlotEnergyCharge	E	Y	Y	N	Levels	Y	Y	Y	N
Fused DO Slot Flash	HUDFusedDOSlotFlash	E	Y	N	Y	Levels	Y	N	Y	N
DO Slot Shatter	HUDDOSlotShatter	E	Y	N	Y	Levels	Y	Y	Y	N
Low Health Indicator	HUDLowHealthIndicator	E	Y	Y	Y	Levels	Y	Y	Y	N
Money Tick Up	HUDMoneyTickUp	E	Y	N	Y	Levels	Y	Y	Y	N
Money Tick Down	HUDMoneyTickDown	E	Y	N	Y	Levels	Y	Y	Y	N
Money Tick Pulse	HUDMoneyTickPulse	E	Y	N	Y	Levels	Y	Y	Y	N
Money Tick End (Cha-Ching!)	HUDMoneyTickEnd	E	Y	N	Y	Levels	Y	Y	Y	N
Particles and Effects										
Explosion	Explosion	E	Y	N	N	Levels	Y	Y	Y	N
Fire	Fire	E	Y	Y	N	Levels	Y	Y	Y	N
Ember Burst	EmberBurst	E	Y	N	N	Levels	Y	Y	Y	N
Firewall Fizzle	FirewallFizzle	E	Y	N	N	Levels	Y	Y	Y	N
Energy Seep	EnergySeep	E	Y	Y	N	Levels	Y	Y	Y	N
Lightning Crackle	LightningCrackle	E	Y	N	Y	Levels	Y	Y	Y	N
Lightning Small Strike	LightningSmallStrike	E	Y	N	Y	Levels	Y	Y	Y	N
Lightning Large Strike	LightningLargeStrike	E	Y	N	Y	Levels	Y	Y	Y	N
Lightning Crash	LightningCrash	E	Y	N	Y	Levels	Y	Y	Y	N
Energy Shine	EnergyShine	E	Y	N	N	Levels	Y	Y	Y	N
Energy Swirl (CircularEnergyTrail)	EnergySwirl	E	Y	Y	N	Levels	Y	Y	Y	N
Spark	Spark	E	Y	Y	N	Levels	Y	Y	Y	N
Energy Charge 1	EnergyCharge1	E	Y	Y	N	Levels	Y	Y	Y	N
Cannon Fire 1	CannonFire1	E	Y	N	Y	Levels	Y	Y	Y	N
Energy Bullet	EnergyBullet	E	Y	Y	Y	Levels	Y	Y	Y	N
Energy Bullet Impact	EnergyBulletImpact	E	Y	N	Y	Levels	Y	Y	Y	N
Skidding	Skidding	E	Y	Y	Y	Levels	Y	N	Y	N
SeeingStars	SeeingStars	E	Y	Y	N	Levels	Y	Y	Y	N
Air Rush	AirRush	E	Y	Y	Y	Levels	Y	Y	Y	N
Chain	ChainClick	E	Y	N	Y	Levels	Y	Y	Y	N
Misc.										
Screen Glitch Sound	ScreenGlitch	E	Y	N	Y	Levels	Y	Y	Y	N
World Spawn	WorldSpawn	E	Y	Y	Y	Levels	Y	N	Y	N
World Spawn Expand	WorldSpawnExpand	E	N	Y	Y	Levels	N	N	N	N
Sky Hexagon Transition	SkyHexagonTransition	E	Y	N	Y	Levels	N	N	N	N
Sky Hexagon Red Ambient	SkyHexagonGlitch	E	Y	Y	Y	Levels	Y	Y	Y	N
Camera Quake					N					

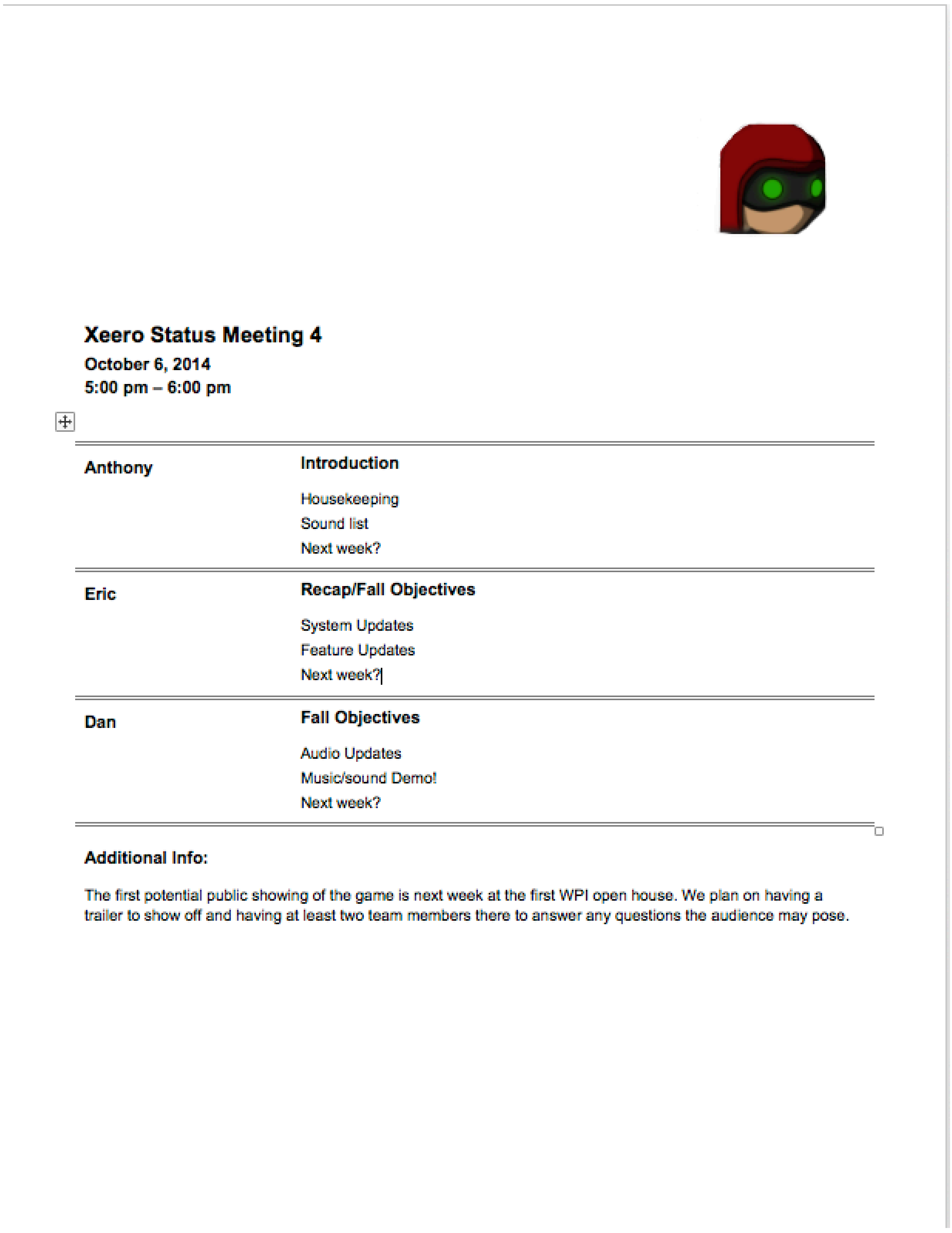
Look Into Options	Temp Completed	124
Needs to be created, high priority	Total	192
Needs to be updated		
Needs to be created, but low priority		

Appendix D: Sample Gantt Chart



		Name	Duration	Start	
1		Open House 1	1 day	10/13/14 8:00 AM	10/
2		Open House 2	0.875 days	11/11/14 9:00 AM	11/
3		System Implimentation	71 days	9/8/14 8:00 AM	12/
4		Feature Implimentation	71 days	9/8/14 8:00 AM	12/
5		DGD Test Day	0.5 days	11/4/14 1:00 PM	11/
6		Demo Music	21 days	9/8/14 8:00 AM	10/
7		Open House Trailer	5 days	10/7/14 8:00 AM	10/
8		Demo Sounds Complete	21 days	9/8/14 8:00 AM	10/
9		Audio Work	46 days	10/13/14 8:00 AM	12/

Appendix E: Sample Meeting Minutes



Xeero Status Meeting 4

October 6, 2014

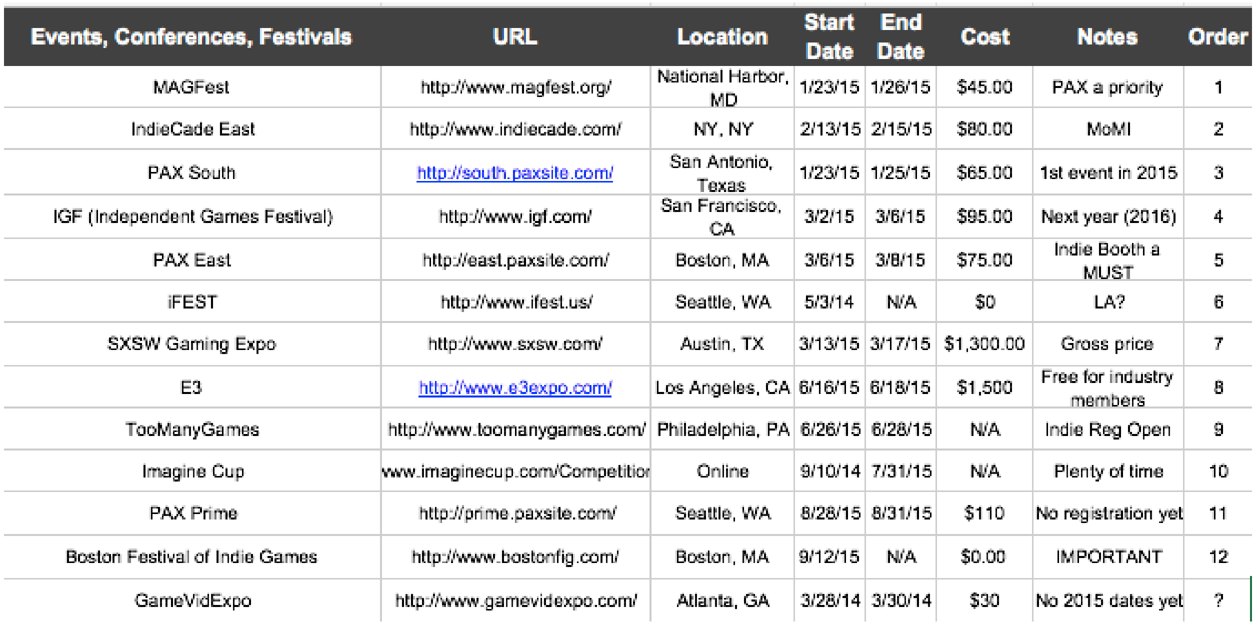
5:00 pm – 6:00 pm

Anthony	**Introduction** Housekeeping Sound list Next week?
Eric	**Recap/Fall Objectives** System Updates Feature Updates Next week?
Dan	**Fall Objectives** Audio Updates Music/sound Demo! Next week?

Additional Info:

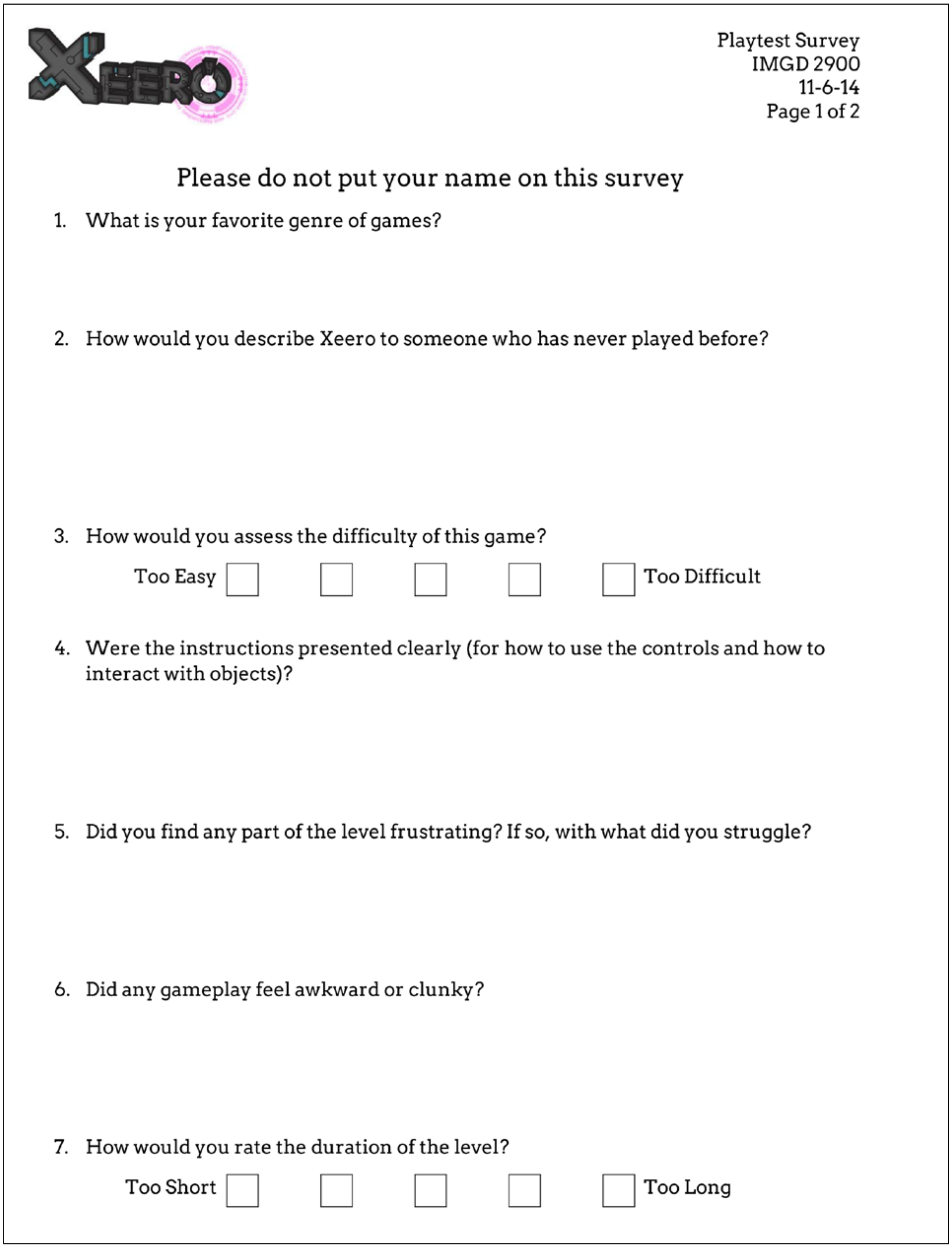
The first potential public showing of the game is next week at the first WPI open house. We plan on having a trailer to show off and having at least two team members there to answer any questions the audience may pose.

Appendix F: Festival Schedule



Events, Conferences, Festivals	URL	Location	Start Date	End Date	Cost	Notes	Order
MAGFest	http://www.magfest.org/	National Harbor, MD	1/23/15	1/26/15	$45.00	PAX a priority	1
IndieCade East	http://www.indiecade.com/	NY, NY	2/13/15	2/15/15	$80.00	MoMI	2
PAX South	http://south.paxsite.com/	San Antonio, Texas	1/23/15	1/25/15	$65.00	1st event in 2015	3
IGF (Independent Games Festival)	http://www.igf.com/	San Francisco, CA	3/2/15	3/6/15	$95.00	Next year (2016)	4
PAX East	http://east.paxsite.com/	Boston, MA	3/6/15	3/8/15	$75.00	Indie Booth a MUST	5
iFEST	http://www.ifest.us/	Seattle, WA	5/3/14	N/A	$0	LA?	6
SXSW Gaming Expo	http://www.sxsw.com/	Austin, TX	3/13/15	3/17/15	$1,300.00	Gross price	7
E3	http://www.e3expo.com/	Los Angeles, CA	6/16/15	6/18/15	$1,500	Free for industry members	8
TooManyGames	http://www.toomanygames.com/	Philadelphia, PA	6/26/15	6/28/15	N/A	Indie Reg Open	9
Imagine Cup	www.imaginecup.com/Competitior	Online	9/10/14	7/31/15	N/A	Plenty of time	10
PAX Prime	http://prime.paxsite.com/	Seattle, WA	8/28/15	8/31/15	$110	No registration yet	11
Boston Festival of Indie Games	http://www.bostonfig.com/	Boston, MA	9/12/15	N/A	$0.00	IMPORTANT	12
GameVidExpo	http://www.gamevidexpo.com/	Atlanta, GA	3/28/14	3/30/14	$30	No 2015 dates yet	?

Appendix G: *Xeero* Test Protocol #1 – Post-Play Survey



Playtest Survey
IMGD 2900
11-6-14
Page 1 of 2

Please do not put your name on this survey

1. What is your favorite genre of games?

2. How would you describe Xeero to someone who has never played before?

3. How would you assess the difficulty of this game?

   Too Easy ☐ ☐ ☐ ☐ ☐ Too Difficult

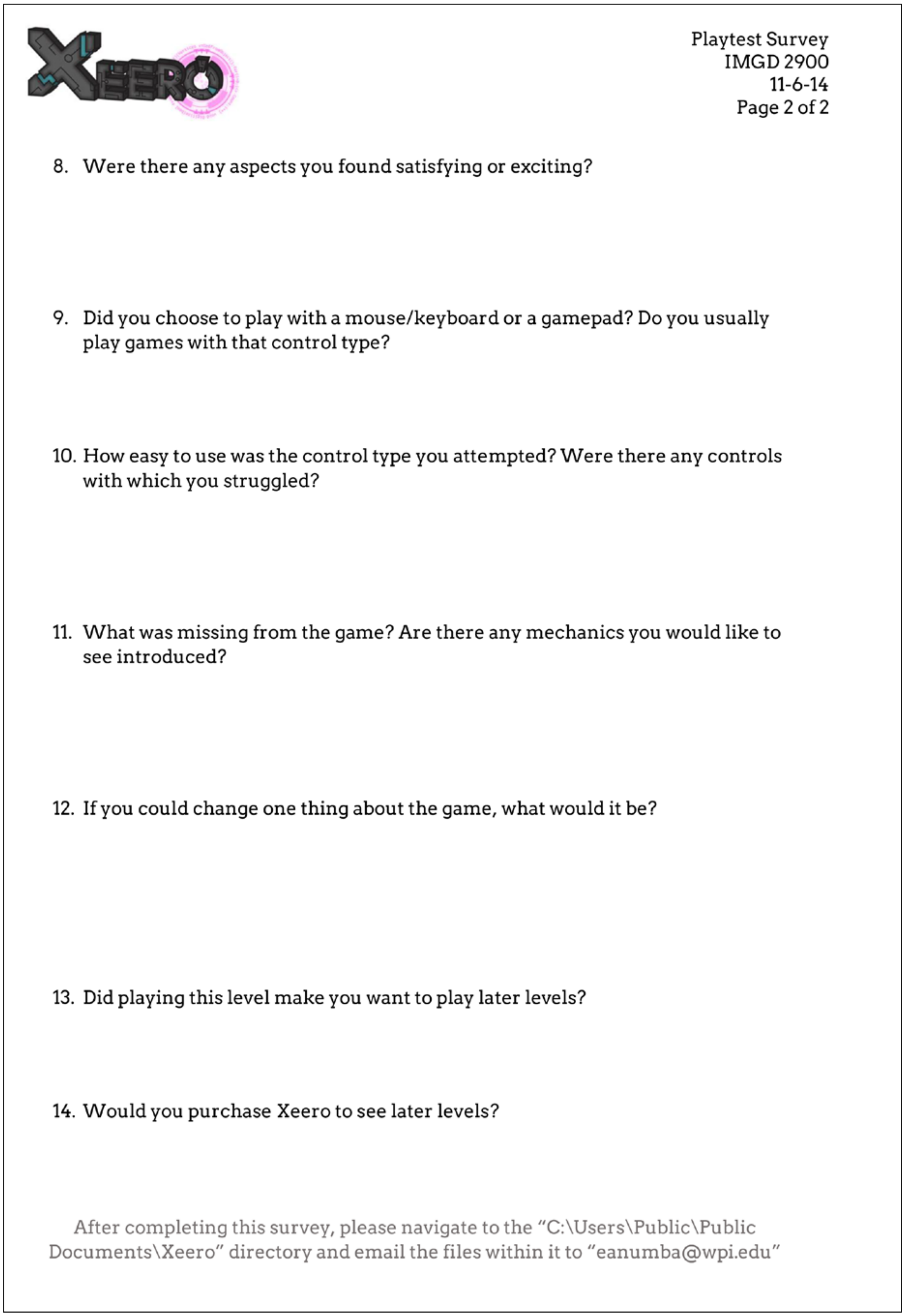
4. Were the instructions presented clearly (for how to use the controls and how to interact with objects)?

5. Did you find any part of the level frustrating? If so, with what did you struggle?

6. Did any gameplay feel awkward or clunky?

7. How would you rate the duration of the level?

   Too Short ☐ ☐ ☐ ☐ ☐ Too Long



Playtest Survey
IMGD 2900
11-6-14
Page 2 of 2

8. Were there any aspects you found satisfying or exciting?

9. Did you choose to play with a mouse/keyboard or a gamepad? Do you usually play games with that control type?

10. How easy to use was the control type you attempted? Were there any controls with which you struggled?

11. What was missing from the game? Are there any mechanics you would like to see introduced?

12. If you could change one thing about the game, what would it be?

13. Did playing this level make you want to play later levels?

14. Would you purchase Xeero to see later levels?

After completing this survey, please navigate to the "C:\Users\Public\Public Documents\Xeero" directory and email the files within it to "eanumba@wpi.edu"

Appendix H: *Xeero* Test Protocol #2 – Overview

Testing Protocol for Xeero

Iteration 3
IMGD 5400
Eric Anumba, Anthony Sessa, Dan Acito, Dan Manzo, Caitlin Malone

Testing summary

These tests are designed for the 3D action-puzzle-platformer Xeero in order to evaluate the clarity of instruction, pacing of levels, and overall difficulty. These tests are to be conducted using playtesters who are new to the game in order to evaluate their initial reactions and their ability to complete the first few levels.

The following further details the aspect of the game are to be tested:

Instruction

- How clear are instructions/tutorial messages?
- Can they describe how mechanics work after they have been presented with the instruction?
- When does it click/what would they have liked to have seen when they were learning it?
- Are objectives within levels clear? Do they find themselves getting stuck anywhere?

Level Pacing

- Are levels too long? Do they feel like they need breaks *within* levels?
- How do they find the length of individual encounters? Any particular encounters that feel too long or short?

Difficulty

- Do they find themselves retrying jumps “too often?”
- Do they find themselves frequently retrying combat encounters?
- Are the checkpoints spaced close enough so players can quickly get back to retrying the area at which they failed?

Protocol Details

Materials Needed & Controls

- In-game questionnaire: This is to be filled out by a data collector who watched the players move through the levels and asked questions as they reached key points in the level to gauge their impressions about specific game details.
- Post-play survey: This is to be filled out by the player after the complete the game to gauge their overall impressions with the game.
- Post-play video data: As the players play the game, the game will collect stats and record a video of the player moving through the levels to be reviewed after the test has concluded.
- Location: All test players will play in the IMGD Zoo lab.
- Testing machines: The computers needed are already present in the Zoo lab, and are Windows 7 machines. No outside software will be used during our tests. The players have the option of using a keyboard and mouse for input, or an Xbox controller (connected to the machine by USB). *Note: We had access to only one Xbox controller. In*

addition, we did not have access to headphones and the computers in the Zoo lab do not have speakers, so sound was not evaluated.

- Build: All playtesters are provided the same build of the game.
- Food: All playtesters will be rewarded with pizza.
- People: Testing can't be conducted without playtesters.

Game Content

This protocol tests the tutorial level (Program 1-1), the Hub (an overworld level), and the second level (Program 1-2), comprising about 40-60 minutes of gameplay.

Test Walkthrough

1. Launch game build on lab machine
2. Playtester chooses control scheme (keyboard or gamepad)
3. Playtester plays through Program 1-1, the Hub, and Program 1-2

a. A team member administers the in-game questionnaire as the playtester plays

4. Playtester fills out post-play survey (scale and short answer)

. A team member points the tester to the survey upon completion of the test

5. Playtester eats free food
6. Team reviews gameplay videos

. The game build records all play sessions so further review can be conducted later on

In-Game Questionnaire

Levels are broken up based on the discrete obstacles they contain. At key points, data collectors observing the players will ask questions about a specific mechanic, combat sequence, or puzzle. Some general types of questions asked include:

- Can players describe the objective of a particular puzzle?
- Can players describe the purpose of a game mechanic?
- Did the puzzle or combat section seem too long, too short, too hard, or too easy?

Post-Play Survey questions

Likert/scaled answers (6 degrees)

1. ***Combat*** - How often did you find yourself retrying combat encounters? / How often did you complete combat encounters on the first try?
2. ***Puzzle solving*** - Are objectives within levels clear to you?
3. ***Instruction Clarity*** - How clear were instructions/tutorial messages?

a. Agree/disagree: Tutorial messages were helpful. / Instructions were clear.

4. ***Level Duration*** - How would you rate the duration of the level?

Open questions

1. What is your favorite genre of games?
2. How would you describe the game to someone who has never played before?
3. ***Platforming*** - Did you find yourself retrying to jump "too often?"
4. ***Puzzle Solving*** - Did you find yourself getting stuck anywhere?

5. ***Combat*** - How did you find the length of individual encounters? Any particular ones that felt too long or short?
6. ***Instruction Clarity*** - Any instructions you didn’t understand?
7. ***Instruction Clarity*** - Can you describe how mechanics work after you were presented with the instructions?
8. Did you want to do something the game wouldn’t let you?

Appendix I: *Xeero* Test Protocol #2 – In-Game Survey

Xeero Playtest

- Encourage the players to talk out loud. What are they trying to do? What are their impressions? Why are they making the decisions they're making?
- The answer to some of these questions may be incredibly obvious to the players, but we're trying to gauge their comprehension.
- Remind them that "I don't know" is an acceptable answer to any question.

Tutorial (Program 1-1)

The majority of the questions here are about gauging the player's understanding for the rules of the game, and to see how effective the tutorial messages are at explaining them.

First Breakpoint

After seeing the info page for the breakpoint and touching it, ask the player to explain the purpose of the breakpoint. Can they describe how they think it works?

First Long Jump

How many attempts did they make before they crossed the long gap?

Ask the player to explain the process of crossing a large gap like that if they encounter it again? (The purpose to ensure they understand *why* they made the jump. This part is designed to teach them how to time the double jump for maximum distance)

Split Path

Which path did they attempt first? Path with the block or the path to the chip?

If they chose the block path, ask them why (after they reach the chip path). Did they not see the other path or did they choose to ignore it?

Digitizing the Block

After gaining the DigitizeBlock() upgrade, ask the player to explain what they think the tutorial prompt means.

After returning to the platform, ask the player to explain again how they think Digitizing works. Ask them if the prompt helped their understanding of the mechanic.

Double Wide Gap

How many times did they attempt to cross the gap?

How many attempts before they placed the Block below them?

Facing Malware

Did they try to use dodging to avoid Malware?

After they escape the malware ask them: was it immediately obvious that they needed to run?

Dodging Turrets

How many attempts before they passed the turrets? Did they use dodging? Jumping?

Ask them how they would rate the difficulty of that section.

Too Easy	1		2		3		4		5		6	Too Difficult

Hub

Did they try going forward or backwards off of the first island? If backward, ask them why they choose that path.

Did they go left or right? Ask them why.

Program 1-2

These questions are geared towards gauging the difficulty and pacing of the level.

Second Gate

As the approach the second gate, ask them to explain what they are trying to do? What is their objective at that point?

How many attempts did it take them to complete the combat encounter?

Did they use any of the archives in combat?

Did they use the block object in combat?

How would they rate the difficulty of the encounter?

Too Easy	1		2		3		4		5		6	Too Difficult

How would they rate the length of the encounter?

Too Short	1		2		3		4		5		6	Too Long

Falling Platforms

How many attempts did they make to cross all of the platforms?

How difficult would they rate the section?

Too Easy	1		2		3		4		5		6	Too Difficult

Block Countering

How many attempts did they make at countering before they successfully countered?

Falling Platform Fight

How many attempts did they make?

How would they rate the difficulty of the encounter?

Too Easy	1		2		3		4		5		6	Too Difficult

How would they rate the length of the encounter?

Too Short	1		2		3		4		5		6	Too Long

Did they block or dodge during this encounter?

Did they counter any attacks?

Did they use the block object in combat during this encounter?

Blocking at the Gate

How many attempts did they make before they reached the side with the turrets?

How many times did they fail to counter an attack (by blocking too soon or too late and by being hit)?

Did they attempt the combat encounter or skip it? Did they partially attempt the fight before skipping it or did they die and then decide to skip it?

Linked Hit Switches

After they have activated the spawners, ask them to explain how these switches work. Do they understand that the switches are linked?

Final Puzzle

Did they try going left (and up) or right (and across) first? Ask them why they chose the path they did?

How many times did they attempt to Digitize the Block at the Falling Platform?

Before leaving, did they immediately try to get the block a second time? If so, ask them why.

How many deaths before they attempted to Digitize the Block below themselves at the second switch?

How many deaths before they successfully hit the switch?

Before leaving, did they immediately try to get the Block a second time? If so, ask them why (if they didn't at the previous block).

How would they rate the difficulty of the final encounter?

Too Easy	1		2		3		4		5		6	Too Difficult

How would they rate the duration of the final encounter?

Too Short	1		2		3		4		5		6	Too Long

In combat, did they:

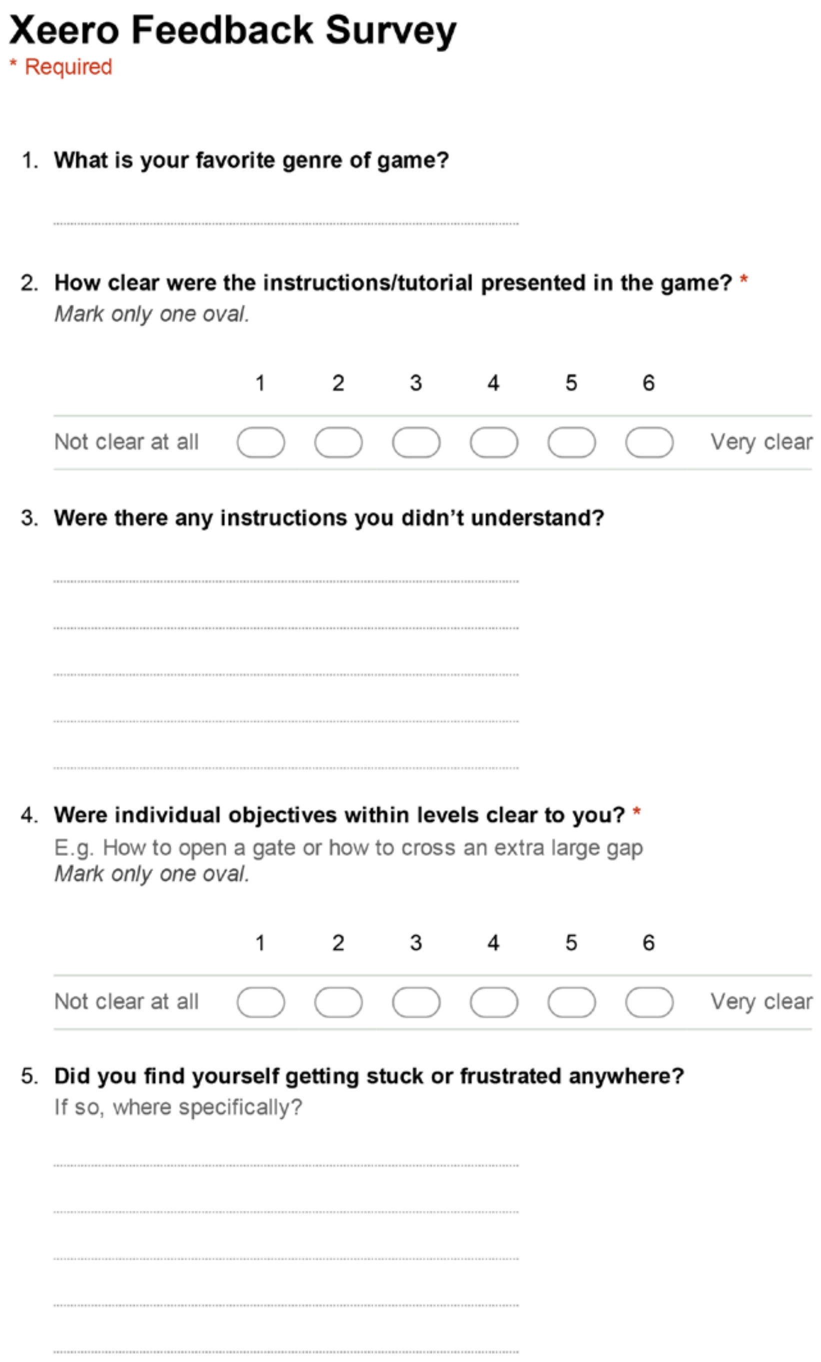
Block		Counter		Dodge		Use the block object	

How many times did they attempt the final encounter?

Ask them if they were developing a strategy to fight the enemies (like going after the bigger ones first, mashing buttons, etc.)?

Extra Notes/Observations

Appendix J: *Xeero* Test Protocol #2 – Post-Play Questionnaire



Xeero Feedback Survey

* Required

1. **What is your favorite genre of game?**

2. **How clear were the instructions/tutorial presented in the game? ***
   Mark only one oval.

	1	2	3	4	5	6	
Not clear at all	⬭	⬭	⬭	⬭	⬭	⬭	Very clear

3. **Were there any instructions you didn't understand?**

4. **Were individual objectives within levels clear to you? ***
   E.g. How to open a gate or how to cross an extra large gap
   Mark only one oval.

	1	2	3	4	5	6	
Not clear at all	⬭	⬭	⬭	⬭	⬭	⬭	Very clear

5. **Did you find yourself getting stuck or frustrated anywhere?**
   If so, where specifically?

6. **How would you rate the overall number of combat encounters?** *
Mark only one oval.

	1	2	3	4	5	6	
Too few	⬭	⬭	⬭	⬭	⬭	⬭	Too many

7. **How would you rate the overall difficulty of the combat encounters?** *
Mark only one oval.

	1	2	3	4	5	6	
Too easy	⬭	⬭	⬭	⬭	⬭	⬭	Too hard

8. **How would you rate the difficulty of making jumps and reaching platforms?** *
Mark only one oval.

	1	2	3	4	5	6	
Too easy	⬭	⬭	⬭	⬭	⬭	⬭	Too hard

9. **How would you rate the duration of the levels within the game?** *
Mark only one oval.

	1	2	3	4	5	6	
Too short	⬭	⬭	⬭	⬭	⬭	⬭	Too long

10. **How would you rate the overall difficulty of the game?** *
Mark only one oval.

	1	2	3	4	5	6	
Too easy	⬭	⬭	⬭	⬭	⬭	⬭	Too hard

11. **Did you ever want to do something the game wouldn't let you?**