

Game Design Document



Viking Jack Games

Jamie Cannon
Steve Chester
Joe Leigh
Ernesto Rojo
Josh Stefanski

confusionlove@gmail.com
chester.steve@gmail.com
urielxvi@hotmail.com
erojo1@fullsail.edu
nikaji@defunctenigma.com

Table of Contents

Game Charter	1
Vision Statement	1
Meeting Schedule	1
Hours Worked per Week	1
When Things Go Wrong	2
Decision-Making Process	2
Rules of Conduct.....	2
Team Roles.....	3
Administrative Roles.....	3
Technical Roles.....	3
Executive Summary	5
High Concept	5
Locale.....	5
Genre.....	5
Basic Controls.....	5
Game Goal	6
Target Platform	7
Marketing & Target Audience.....	7
Game Walkthrough/Overview.....	8
Key Features.....	9
Comparative Products	9
How this Product Stacks up	9
Treatment	11
Dust Jacket Story	11
Game Story.....	11
Interactivity	13
Goal.....	13
Interface	13
Interactive Rhythm.....	16
How the Player Marks Progress.....	16
Detailed Design Breakdown	17
Front End Flow Chart.....	17
Game Flow Chart	18
Glossary of Terms	19
Characters	20
Weapons	28
Power-ups.....	34
Miscellaneous Objects	40
Levels and Maps	44
Combat System	49

Table of Contents

Customization Systems	51
Game Logic, Algorithms, and Rules	52
Key Game Algorithms.....	52
FAQ.....	53
Reference of Key Elements.....	54
Scoring	54
Winning/Losing	54
Rewards	55
Art and Production Design.....	56
Art & Animation Style.....	56
Sound Effects Deliverables	56
Music Deliverables	56
Technical Overview	57
Coding Standards	58
Classes.....	59
Structures	60
Commenting	61
Data Alignment.....	62
Coding Guidelines	63
Lua Exposure	64
Development Environment.....	67
Timing Specifications.....	68
System Feature Breakdown	69
Module Breakdowns	69
Rendering Engine	69
Scene Graph.....	71
Model Loader	72
Model Manager	73
Texture Manager	74
Animation System	75
Collision	76
Camera	77
Input.....	78
Sound Manager	79
Particle System	80
HUD	81
Memory Manager	82
Scripting	83
Kernel.....	84
GlobalTimer.....	85

Table of Contents

Shaders	86
Game Feature Breakdown	87
Prioritized Feature List.....	87
Module Breakdowns	89
Weapon Manager	89
AI Manager.....	91
Projectile Manager	93
Orb Manager	94
Level Manager	95
Menu Manager	96
System Architecture.....	97
Module-specific relationships	100
Utilization Relationships	100
Bidirectional Utilization.....	102
Scripting Interfaces	102
Kernel Tasks	103
Appendix A	105
Memory Map	105
Integration Plan.....	106
Testing Plans	108
Appendix B.....	110
Game Folder Hierarchy.....	110

Game Charter

Vision Statement

Using our skills learned at Full Sail, we plan to challenge ourselves and make a fun and exciting game. We want to fully understand and master the team dynamic and maintain a fun atmosphere with no cramming as we develop a game that will be enjoyable to all types of gamers. We also want our game to involve a unique storyline and setting to enhance a tried-and-true style, allowing simplistic play to appeal across the market. But most of all we want to create something that will allow us each to further our careers in the game industry after graduation by having a polished finished product to show for all our hard work and dedication.

Meeting Schedule

1. For the first two months, we will meet at least three times a week. If we meet on a day we also have class, the meeting will last no more than 4 hours. For the remaining months, we will scale our number of meetings to account for time previously spent in class. No single meeting will last more than 8 hours.
2. Meetings will take place at Joe's house. We will meet in the early afternoon, or occasionally evenings, depending on the length of the meeting and the specific situation.
3. At the beginning of the meeting, each member will discuss any difficulties or problems that may have arisen since the previous meeting. After that, we will plan the meeting tasks, either individual or team efforts, and how long we will work on these tasks. We will take short breaks periodically, at increments determined on a case-specific basis.

Hours Worked per Week

1. Each member will be expected to work at least 35 hours a week, including lab time.

2. The maximum number of hours of assigned work an individual will face is 50 a week. More than this may cause problems or extreme cases of stress.

When Things Go Wrong

1. Group members will inform each other of emergencies by phone calls, text messages, or instant messaging.
2. If something goes wrong for an individual, they should not attempt to solve the problem by themselves. Instead, inform the rest of the group, or the member whose role is most appropriate for the situation, and if a meeting is needed, one will be scheduled immediately.
3. We will reduce the risks of emergencies by keeping high levels of communication, as well as frequently integrating code to avoid overwriting each other's changes.

Decision-Making Process

1. Decisions regarding design or implementation will be made by presenting the problem to every group member and agreeing on a popular consensus.
2. Most situations will be concluded within an hour. However, if emotional stress levels are high and no agreement can be made in this hour, we will agree to end the meeting without a decision and let each member consider the problem further.
3. If no consensus can be reached on a given problem, we will ask an outside source for an objective view of the situation.
4. If the minority of the vote is unsatisfied with the consensus, we will also present the issue to an outside source to make sure the idea has been thoroughly considered.

Rules of Conduct

1. Any member who cannot attend a meeting must inform at least one other member before the meeting begins.
2. Any member who is running more than 10 minutes late from a meeting must inform at least one other member of what time they will arrive.
3. The group will take a break from working periodically to help the members stay on task for longer periods of time.

4. If someone misses a meeting, they are automatically voted to provide food and drink for the next meeting.
5. Any time someone is off task for over ten minutes, it will be documented by the group as an “offense.” The list of offenses will be posted on the forums for the group to see. If someone accumulates more than four offenses in a single month, or if an offense is so distracting as to cause the meeting to end early or any other extreme situation, we will consult our External Producer to decide a suitable action.
 - a. “Off Task” refers to anything not declared at the beginning of a meeting as a task that will be done.
6. Offenses will also include arguments and personal insults. We have a strict no-tolerance policy in this regard and one offense per meeting is the absolute limit of tolerance. Any member who suffers from said offense has the right to bring it to the External Producer’s attention immediately.

Team Roles

Administrative Roles

- **Project Lead** – Jamie Cannon
- **Asset Lead** – Joe Leigh
- **Technical Lead** – Josh Stefanski
- **Gameplay Lead** – Steve Chester
- **Interface Lead** – Ernesto Rojo

Technical Roles

- **HUD & Menus** – Ernesto Rojo
- **Particles** – Joe Leigh
- **Artificial Intelligence** – Steve Chester and Jamie Cannon
- **Animations** – Josh Stefanski
- **Graphics and Rendering** – Josh Stefanski and Joe Leigh
- **Engine Core** – Josh Stefanski
- **Scripting Engine** – Josh Stefanski
- **Sound and Music** – Ernesto Rojo

- **Physics and Collision** – Steve Chester
- **Level Design and Layout** – Jamie Cannon
- **Gameplay** – Jamie Cannon and Steve Chester
- **Texture Managing** – Joe Leigh
- **Memory Manager** – Josh Stefanski
- **Input** – Josh Stefanski and Joe Leigh

Executive Summary

High Concept

It is the year 2138... mankind's reach encompasses nearly the whole solar system... five colonies thrive, yet one force threatens to overwhelm them all... What is the secret of Ziggurat Six?

In an epic struggle for control of the galaxy's power, grab your plasma gun and blow away hordes of supernatural monsters that threaten the stability of the colonies!

Locale

Ziggurat Six takes place over one hundred years in the future. Mankind has colonized our solar system in six places, each powered by a giant temple-like ziggurat. The game takes place in some of these ziggurats. They are constructed partly of super-strong metals, and partly of enormous slabs of stone and concrete, but the interior sections are entirely metal.

- **Mars** – The geometry of this level is mixed between the ground floor and a second higher level that is made up of separate, distinct platforms. The player can reach these platforms with their predefined paths (ramps). The ground floor is separated into a kind of rooms, with open ceilings above them. This is the oldest Ziggurat and therefore is the smallest in the solar system.
- **Titan** – Three-story level. This is more like scaling the inside of a complete Ziggurat, with three large areas, each higher than the previous. Large ramps and platforms can be seen stretching high overhead. Flying gryphons circle over the largest open space in the level. This is the furthest Ziggurat from the sun, therefore requiring more energy drawn from Titan itself in order to create a hospitable environment.

Genre

Action/Shooter/Arcade

Basic Controls

Keyboard and Mouse

Game Goal

Overall Goal:

Defend the solar system by killing as many invading aliens as you can!

Individual level goals:

Survive as long as possible by picking up time orbs and upgrading your weapons strategically.

Get the high score by killing lots of enemies and surviving a long period of time.

Advancement:

Navigate dangerous, trap-filled areas.

Improve your weapons by accumulating items from defeated enemies.

Target Platform

PC

Marketing & Target Audience

There are several key statistics that support our belief that this game will be easy to market and sell:

- Action games constituted 30% of video game sales in 2005.
- The T for Teen rating was the second-highest selling in 2005.
- Nearly half of the 20 top selling games of 2005 fit into the action genre.

Primary Audience

- Males aged 15-30
- Fans of third-person shooters
- Fans of casual, fast-paced, score-based games

Secondary Audience

- Females aged 15-30
- Science fiction fans; viewers of *Battlestar Galactica*, *Stargate SG-1*, *Farscape*, *Firefly*

Game Walkthrough/Overview

Upon starting a new game, you find yourself choosing which Ziggurat you would like to defend. As you orient yourself to either the Mars or Titan Ziggurat, you notice two things: first, the flying enemies circling around the level ahead of you, and secondly, the enemies moving on the ground in the darkness that seem to be coming closer.

As these walking creatures near, you can see their eyes glowing. And as they emerge from darkness you can also tell that they are quadrupeds that take a likeness to the Egyptian god Anubis, and they do indeed seem rather eager to expedite your trip to the Underworld. The scene erupts into a flurry of plasma bolts and sparks, as you dive toward the nearest ramp, frantically spraying the room with cover fire. You catch a hit in the leg, Ow that stings! But the shooter fares much worse, taking your plasma bolts to the chest, instantly dying and dropping a health orb. After the orb absorbs into you, your pain vanishes and you feel right as rain, so you dive back into the fray. Moments later the firefight is over. Thankfully the cyborg Anubis guards weren't nearly as tough as they looked, but you're still a bit shaken up from all the action.

In addition to the red health orb that helped restore you, also a blue and yellow orb spawned from the enemy's corpse. The blue orb immediately gives you more time to survive, and the yellow orb adds to a total that will upgrade your weapons. All in due time, though – you need more orbs for now.

There's not much time to relax, however. Not only are the flying Gryphon-like creatures still circling ahead, but you can see more Anubis have figured out how to get to you. You move forward quickly off your platform and tentatively explore the level.

Cautiously, you advance forward. More cybernetic-enhanced Anubis figures advance toward you, guns firing rapidly. Now you're getting the hang of it as you dispatch them easily, absorbing health orbs and again noticing they also drop a different type of orbs. For now, though, your immediate concern is the Gryphon swooping down from above. It starts shooting fireballs at you and man, do they hurt! You strafe and dodge, then suddenly the word DANGER begins scrolling on the bottom of the screen. What? Without realizing at first you have walked into a trap! The turret was sitting in a dark corner... very tricky. You stay away as you dispatch of the Gryphon, but soon enough more are coming. And the Anubis are after you too.

You flee to a higher platform to pick off the enemies as they approach one at a time. More orbs scatter and then chase toward you, restoring your health, increasing your timer, and filling a yellow bar on the HUD. The alert on the

bottom of the screen appears suddenly, telling you there is an Upgrade Available. Interested, you pause the game with a keystroke and enter the upgrade menu. Here, you discover that the yellow orbs occasionally dropped by enemies have added up, and now you have Upgrade Points to spend. After adding points to the weapon of your choice, you return to the fray and keep racking up the points...

Key Features

- General Features
 - Futuristic setting with elements of mythology
 - Travel to multiple worlds and environments
- Player Upgrades
 - Upgrade your weapons' rate of fire, clip size, etc.
- Gameplay
 - Third-person, fast-paced, gun combat
 - Enemies drop health, time, and experience orbs which are automatically picked up

Comparative Products

Bullet Witch

Devil May Cry

Enter the Matrix

Final Fantasy VII: Dirge of Cerberus

Gears of War

Gungrave

Earth Defense Force 2017

Lost Planet

Ratchet & Clank

Star Wars: Battlefront

Transformers

Van Helsing

How this Product Stacks up

Our game is similar to these products in that they're all third person shooters. Some, like *Gungrave* and *Van Helsing*, are fantasy based. Others, such as

Gears of War and *Ratchet & Clank*, are sci-fi based. We're using orbs to restore health, much the same as *Devil May Cry*. The difference is that the orbs will come to you, instead of having to run over them. The gameplay in *Ziggurat Six* will be fast paced, like in *Enter the Matrix*.

Treatment

Dust Jacket Story

In the year 2138, mankind has expanded to plant six colonies throughout the solar system. Colonies powered by giant temple-like ziggurats that support hundreds of thousands of lives by drawing power from the gods above. Colonies that have thrived for decades until the fateful day horrific monsters start pouring forth in ziggurats around the solar system.

Once a guard protecting Mars' Ziggurat Two, you must now take on these monsters and defend the lifeblood of your home colony from wave after wave of bizarre cyborg creatures. You must travel between the besieged ziggurats, annihilating every monster in your path to discover what evil being is behind these attacks.

Game Story

It's the year 2138, and human civilization has only begun to touch the depths of space exploration. Nearly a century ago, mankind organized into the Earthwide Alliance and began focusing their efforts on colonizing their solar system. When the Colonial Alliance first started colonizing other terrestrial bodies, they were forced by technological limitations to find a new way to live off the land. This innovative research led to the discovery of a new type of energy. This new energy discovered is a result of the buildup of different types of forces exerted on the planets and moons as they travel through space. It is more commonly known as the planet's potential energy and was given the name Terrestrial Energy, or TE. For the last 50 years, technology has given people the ability to absorb this energy for many purposes. The discovery of TE led to the development of new state-of-the-art weapons that harness the energy to provide a revolutionary kind of power.

TE was first perceived to be the "power of the gods" by many people; this idea took hold as scientists discovered an ancient ziggurat was the best place to harvest it more massively. This major breakthrough led to the construction of more technologically advanced ziggurats on Mars and four moons: Europa, Ganymede, Titan, and Titan. TE also helped increase the ability to research quantum mechanics more efficiently. Soon, quantum computers with teleportation support were placed in each ziggurat.

There are special force groups funded by the Colonial Alliance that patrol each ziggurat and make sure nothing goes wrong. The main concern of the Colonial Alliance is of activist groups who believe TE is not a gift from the heavens, but instead a vital essence of every terrestrial object. These groups are widely labeled as terrorists for continual attempts at destroying the TE-harvesting ziggurats. The members of the Ziggurat Protection Squads are highly trained individuals and have a thorough knowledge of TE.

Thorough Colonial Alliance security crackdowns limiting the actions of its citizens have put a stop to these terrorist attacks for many years. Therefore, security has been lowered on site considerably as the risk of attack is substantially lower than in years past. The most recent ziggurat was codenamed Ziggurat Six since it is the sixth ziggurat and is still under construction on Neptune's moon, Titan. Scientists have already completed the installation of a quantum computer in Ziggurat 6 that can speed up the process by teleporting construction workers to the site.

Interactivity

Goal

The main goal of the player is to destroy as many enemies as possible before the time runs out. The player can raise his/her timer by collecting time orbs from the enemies they kill.

Interface

HUD:



The HUD is used to display your current health, shield, mechanical points earned, weapon, and the heat status of this weapon.

The heat bar, the leftmost bar on the HUD, is a special indicator that fills as the weapon is fired. As each bullet adds to the current heat of the weapon, every

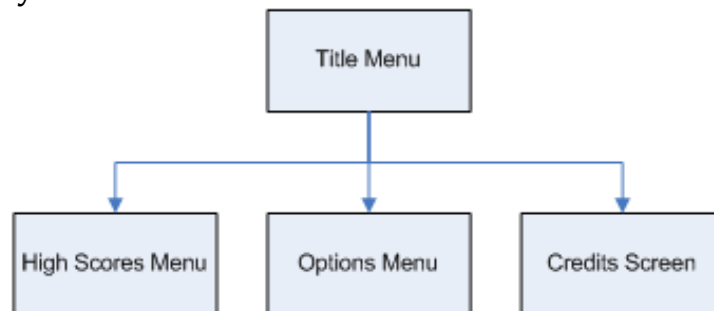
shot will fill the bar with blue until it reaches the overheated status. At this point the overheat bar will be outlined with red.

The next ring, which is light blue in the illustration above, represents your shield value. Every hit you take will lower your shield. Without shield, your health will take damage. Your health is shown in the seven segments inside the shield circle, which are filled with red to show full health. Whenever you take damage, your health segments are outlined in red just like the overheat bar.

Finally, the inner most circle fills its seven segments with gold as you pick up mechanical orbs. When you have enough orbs to gain an upgrade point, the bars on the HUD empty again but are left with a gold outline for as long as you have an upgrade point to spend.

Menus

Menu Hierarchy/Flow:



Title Menu

- Start– Starts a new game.
- Options – Allows you to customize your play experience.
- High Scores – View the high scores.
- Credits – Display the credits screen.
- Exit – Quit the game.

Select Ziggurat

- Mars – Chooses Mars as the level you would like to play.
- Titan – Chooses Titan as the level you would like to play.

Options Menu

- FX – Adjusts the current sound effect volume. Changes take effect immediately and a test sound is played every time.
- Music – Adjusts the current background music volume. Changes take effect immediately.

- Gamma – Adjusts the gamma correction level, changes are displayed immediately.

Pause Menu

- Resume Game – Return to the game in progress.
- Upgrade – Load Weapon Upgrade screen.
- Options – Load Options menu.
- Exit to Menu – Exits the current game and returns to the Title menu screen.

Upgrade Menu (see image below)

- Damage – Increases the damage for any of the three weapons.
- Rate of Fire – Increases the rate of fire for the machine gun.
- Cooldown – Increases the speed at which the pistol or machine gun cools down.
- Left/Right Arrows – Used to select a different weapon to upgrade.
- Back – Returns to the pause menu.



Interactive Rhythm

Typical section durations:

1. Session: 1-2 levels or 3-30 minutes.
2. Level (Ziggurat): 3-15 minutes.
3. Overall game: 3-30 hours.

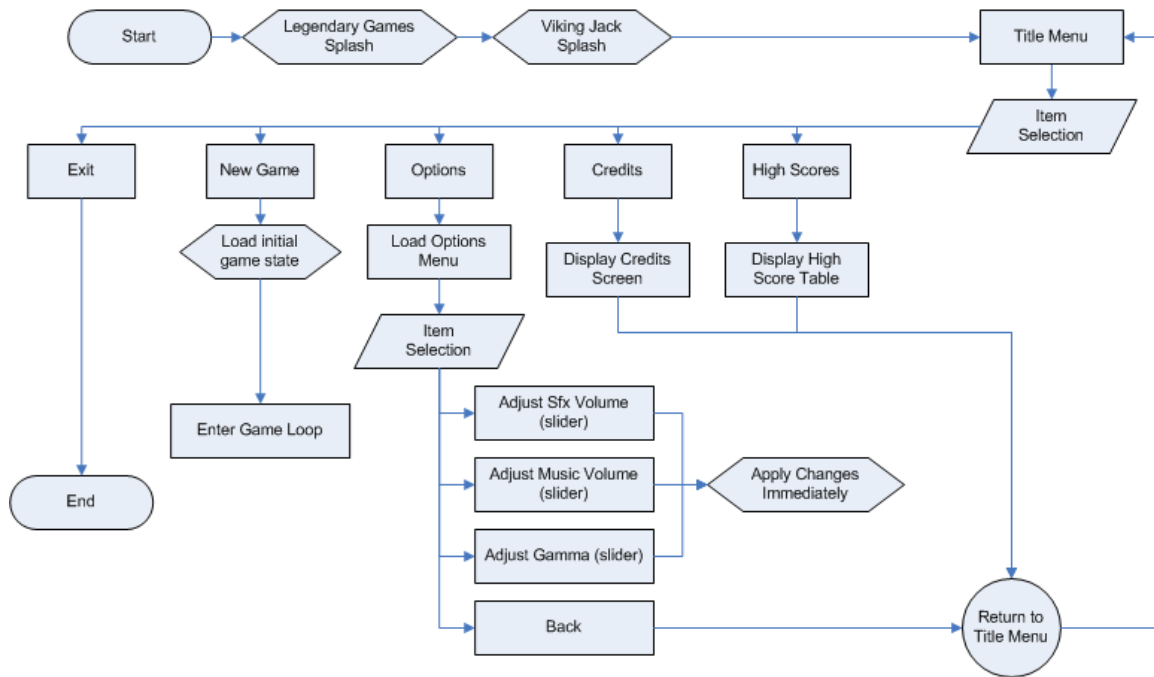
Since the main purpose of the game is to kill as many enemies as possible before the time runs out, there should be very little downtime between enemy encounters. The enemy difficulty in each level gradually increases the longer the player survives. To keep the game from becoming too difficult, the player will have the option to upgrade their weapons as they collect orbs.

How the Player Marks Progress

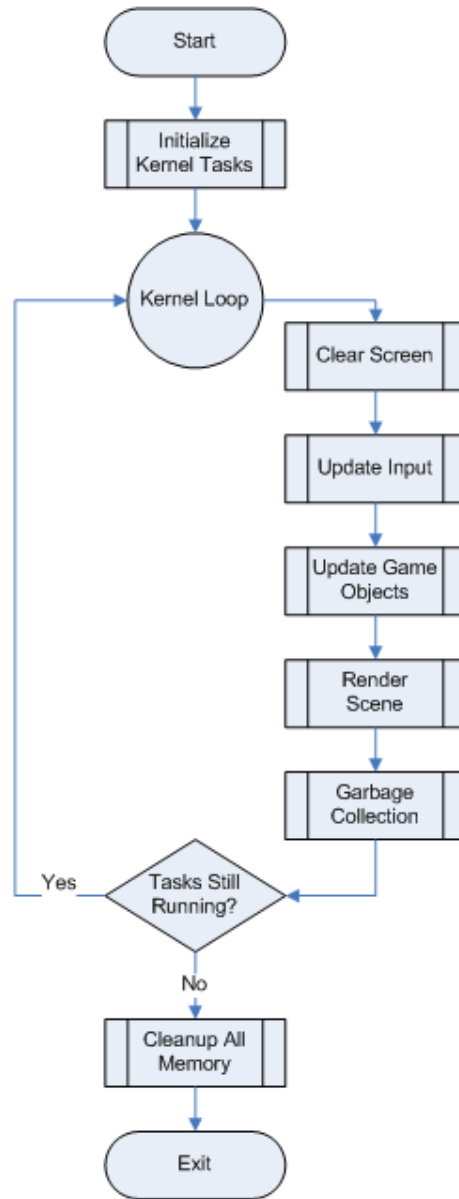
The player will be able to mark their progress in a number of different ways. The main ways the player can do this is by how long they are surviving in a level, how powerful their weapons have become (how many upgrade points spent), or how high their score is. All of these show how long the player has been alive for and whether or not they've defeated a large amount of enemies.

Detailed Design Breakdown

Front End Flow Chart



Game Flow Chart



Glossary of Terms

Blast Radius	The reach of splash damage spreading from the point of impact.
Cool down Rate	The amount of heat lost per second while the gun is cooling down.
Cool down Time	The amount of time the gun must be idle before it will begin to cool down.
Damage	The amount of health removed from the target on impact.
Defense	The amount of damage the player can take before they begin to lose health.
Health	The amount of damage the entity can take before they die/are destroyed.
Health Restored	The amount of the player's health that is returned.
Heat Limit	The maximum amount of heat the weapon can reach before needing to cool down.
Heat per Bullet	The amount of heat gained from each shot fired.
Mechanical Points	Points that eventually grant upgrade points. 100 Mechanical Points = 1 Upgrade Point. Mechanical points are gained by collecting Mechanical Orbs.
Mechanical Points Gained	The amount of mechanical points awarded to the player.
Movement speed	The rate at which the entity travels through the world.
Rate of Fire	How many shots this gun can fire per second.
Shield Recharge	The rate at which the player's defense score recharges.
Upgrade Points	Points that allow upgrading weapons. One upgrade point allows one upgrade that can be spent on any weapon.

Characters

Name/ID

Nimbus / PC

Brief Description

Nimbus is the playable character in Ziggurat Six. This is the avatar taken on by the player in order to progress through the game.

Although Nimbus is human, he is also considered a Martian since he was born and raised on Mars. He stands 2 meters tall and weighs 176 pounds. His armor is designed with a complex set of metals and mechanisms to help protect his body from electromagnetic radiation. His right arm is cybernetic, made of light yet strong metal instead of flesh and bone, colored dark, shiny gray. The metal on his armor is a light gray color and fits tightly to his body. There are embellishments on the armor such as symbols that will be colored to the player's choosing.

Back Story

Nimbus was born in 2102 around the Pindus Mons mountain range on Mars. His father was a TE scientist and worked in Ziggurat Two and his mother stayed at home to raise him. Both his parents died from old age in the '20s. Nimbus, always being around TE, grew a large interest in the subject and hoped to continue his father's work. He was employed by the Colonial Alliance to help defend Ziggurat Two in 2122.

For the last 16 years, Nimbus has spent most of his life inside the ziggurat researching TE and keeping it defended from any terrorist organizations. He now lives in the colony located near Ziggurat Two and hopes to one day explore past Neptune and research the Trans-Neptunian Objects in the Kuiper Belt. He, like many others in his time, believe there are fantastic creatures and powers beyond the reaches of the Sol system.

Behaviors

- Move
- Equip Weapon
- Fire Weapon
-

Attributes

Health	700
Shield	300
Shield Recharge	4 sec without taking damage
Mechanical Points	Begins at 0. Every Mechanical Orb picked up adds one point to the total. When the value reaches 100 it is reset to 0 and the character gains an Upgrade Point.
Upgrade Points	Begins at 0. Every 100 Mechanical Points gives an upgrade point. They can be spent at any time during the game. There is no maximum limit of upgrade points.
Movement speed	2 m/s

Dialogue

Grunt when taking damage

Visual Design



Name/ID

Mech Anubis / Enemy1

Brief Description

Fundamentally this is a human torso with a jackal head and four mechanical spider legs. The upper body is organic, with the tan flesh color and black animal head of classic Egyptian mythology. The lower body is mechanical, creating a cyborg being. Its head and shoulders are adorned with classic Egyptian headdress and jewelry. It carries a rifle weapon that functions the same as the player's two-handed weapon. The model can be the same and the enemy will use the weapon with both hands.

Back Story

From a trans-Neptunian planet thought to be the origin of Earth's Egyptian influences, the Mech Anubis was originally conceived during a period of technological advancement. The Mech Anubis sentry is comprised of an upper torso likeness of the well-known God of death, Anubis, melded with a four-legged cybernetic abdomen. Anubis was chosen to strike fear and a sense of inevitable doom into its enemies because of his role as the guardian of the Underworld. It is believed that when you see the face of Anubis, you've already started your journey to the underworld. The environment on the originating planet is also very dry and desert-covered, so it was decided that the sentry would be a quadruped in order to traverse the sand-covered lands quickly.

Behaviors

- Move
- Fire Gun

Attributes

Health	500
Movement Speed	1.07 m/s
Rate of Fire	0.5 shots/second
Damage	30
Mechanical Points Gained	2-6

Dialogue

Shout or battle cry that triggers when it enters its aggressive state and starts chasing the player.

Visual Design



Name/ID

Mech Gryphon / Enemy2

Brief Description

Gryphons are the flying units of the Mars level. They are weaker in health and damage than the Mech Anubis. They approach until the player is in range of their breath attack, then stay hovering in the air. In form they are a classical Gryphon with a lion body, falcon head, and feathered wings, but their head has some metal parts.

Like the Mech Anubis, this is a hybrid between an organic lifeform and machinery. Basically, it has the body of a lion, large feathered wings, and the head of a falcon. The head is colorized according to Egyptian myth, meaning it is not the natural tans, browns, and white, but instead the colors of a ceremonial headdress. The wings are mechanical whereas the body and head are organic, except the lion's claws should be metal. They are also more vicious and over exaggerated than a regular lion, as suited to mythology. It will attack by breathing fire from its mouth. It is four feet tall at the shoulder.

Back Story

Sharing both the same origin and role as the Mech Anubis, the Mech Sphinx was designed to be the Mech Anubis' aerial counterpart. The Hierocosphinx was chosen as a base because of its falcon head melded with its cybernetically-enhanced flying abilities, in addition to the sphinx being a guardian symbol. The Mech Sphinx was granted cybernetic wings in order for it to easily patrol and watch over the vast desert plains. The Mech Sphinxes were entrusted with guarding only the most precious assets of their society, which lead to the development of their incredibly aggressive personality. It is said that no one has lived after seeing an enraged Mech Sphinx, for if they are enraged, that person has already come too close to what they were guarding.

Behaviors

- Move
- Breathe Fire

Attributes

Health	400
Movement Speed	1.25 m/s
Damage per Second	25 damage/sec
Mechanical Orbs Gained	2-6

Dialogue

When the player comes near enough to trigger its aggressive state and the Mech Sphinx gives chase, it will let out a hawk's cry. The sound should be slightly modified as though to sound technological or coming from a machine.

Visual Design



Weapons

Name/ID

Plasma Pistol / Weapon1

Brief Description

This is the most basic gun used by the character. It does average damage but fires like a handgun – only as often as you pull the trigger. Also it may only shoot ten consecutive bullets before overheating.

It is a one-handed weapon that will be used in the player's right hand. Its basic colors are black, silver, dark blue, basically metallic in structure and color. The plasma bolt it fires will be a bright blue.

Behaviors

- Shoot
- Overheat
- Cool Down

Attributes

Rate of Fire	When button is pressed
Damage	50
Heat per Bullet	10
Heat Limit	100
Cool down Time	2.5 sec
Cool down Rate	10/sec

Visual Design

This gun will closely resemble the plasma assault rifle from the Halo series. The plasma shot will fire out of the very end between the prongs, and the character will hold it from the right most bar.



Name/ID

Plasma Rifle / Weapon 2

Brief Description

This is the secondary gun the player will use. It is similar to a machine gun. It fires fast, but does less damage, it is intended to add some variety and give the player a choice of which gun they prefer.

Behaviors

- Shoot
- Overheat
- Cool Down

Attributes

Rate of Fire	6 shots/second
Damage	20
Heat per Bullet	2
Heat Limit	100
Cool down Time	2.5 sec
Cool down Rate	10/sec

Visual Design

It is a large bulky two-handed gun. There is a butt to the gun, to extend its length, and so the player can shoot it from their shoulder. The core of the gun is a glowing chamber that represents red plasma.



Name/ID

Plasma Cannon / 3

Brief Description

The rocket launcher fires a glowing orb that explodes on impact. It will do damage in a blast radius, and will have a long cool down to prevent over use.

Behaviors

- Shoot
- Overheat
- Cool Down

Attributes

Rate of Fire	0.5 shots/second
Damage	100
Heat per Bullet	100
Heat Limit	100
Cool down Time	2 sec
Cool down Rate	15/sec
Blast Radius	1.0

Visual Design

The rocket launcher is very large and takes over the whole arm. The front of the weapon looks similar to the front of the first weapon.



Power-ups

Name/ID

Health Orb / Powerup1

Brief Description

These items will be small, glowing, red orbs that are dropped by enemies once killed. The player will absorb these orbs on contact. While the player is in combat, the orbs will slowly track towards the player's position, and if the player is out of combat, they will have the option of pressing a button to accelerate the orbs' approach.

Each orb restores a small amount of the player's health.

Behaviors

- Move

Attributes

Health Restored	20
Tracking Speed	Various

Visual Design

This item is a small sphere about the size of the character's hand. It is red and glowing brightly.



Name/ID

Mechanical Orb / Powerup2

Brief Description

These items will be small, glowing, yellow orbs that are occasionally dropped by enemies when killed. The player will absorb these orbs on contact. The orbs will slowly track towards the player's position.

Each orb adds to a numerical score, Mechanical Points, that the player can use to upgrade their equipment.

Behaviors

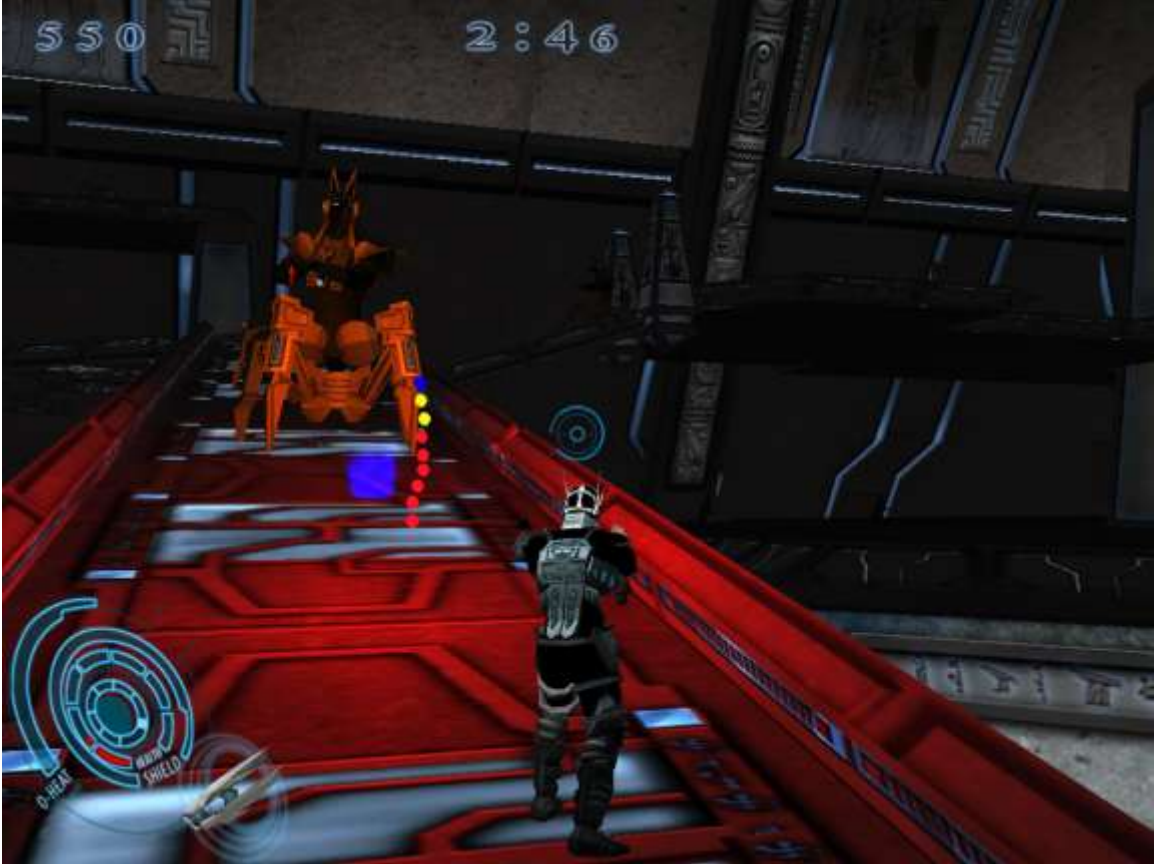
- Move

Attributes

Mechanical Points Gained	1
Tracking Speed	Various

Visual Design

This item is a small sphere about the size of the character's hand. It is yellow and glowing brightly.



Name/ID

Time Orb / Powerup3

Brief Description

These items will be small, glowing, blue orbs that are occasionally dropped by enemies when killed. The player will absorb these orbs on contact. The orbs will slowly track towards the player's position. Each time orb adds to the player's timer.

Behaviors

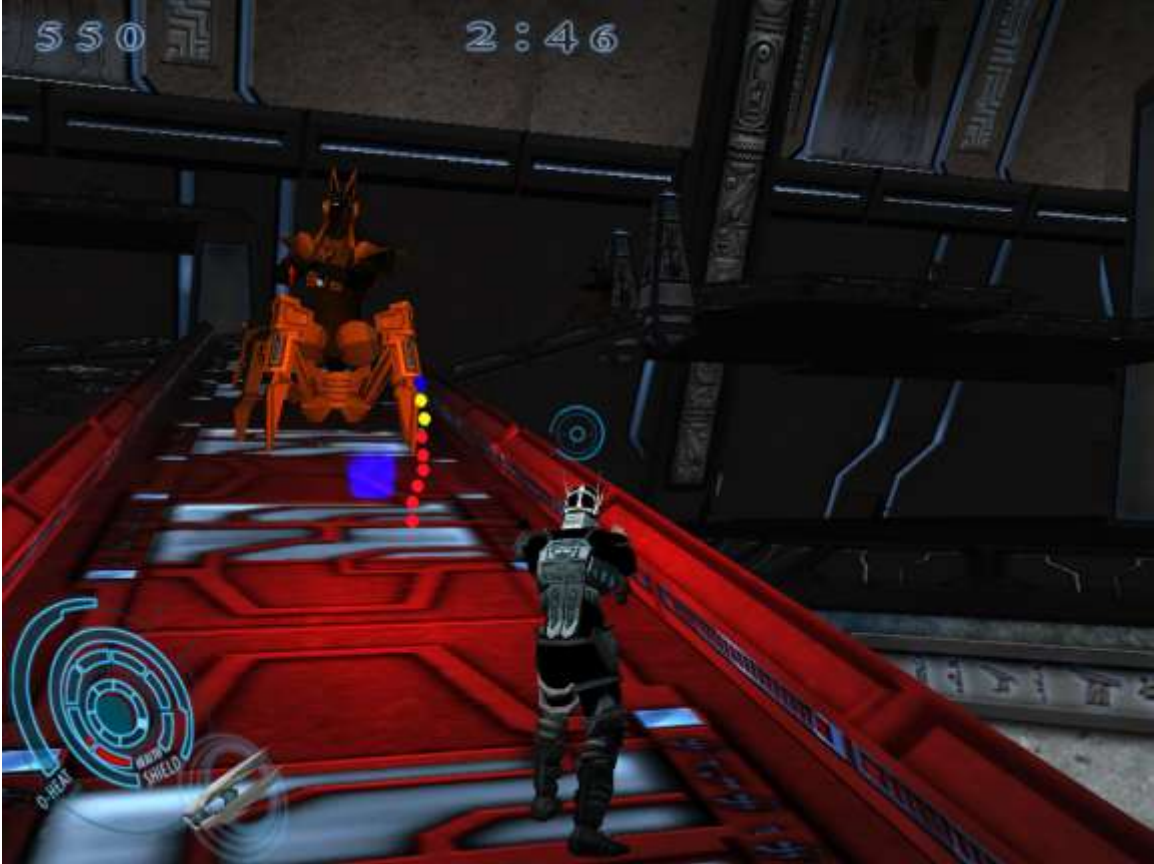
- Move

Attributes

Seconds Gained	10
Tracking Speed	Various

Visual Design

This item is a small sphere about the size of the character's hand. It is blue and glowing brightly.



Miscellaneous Objects

Name/ID

Turret / Object1

Brief Description

Turrets function as a hazard element within the levels. They are stationary weapons planted on the floor in certain rooms. A turret does a significant amount of damage to the player, but not enough damage to kill them in one shot. A console must be activated that disables the turret.

Behaviors

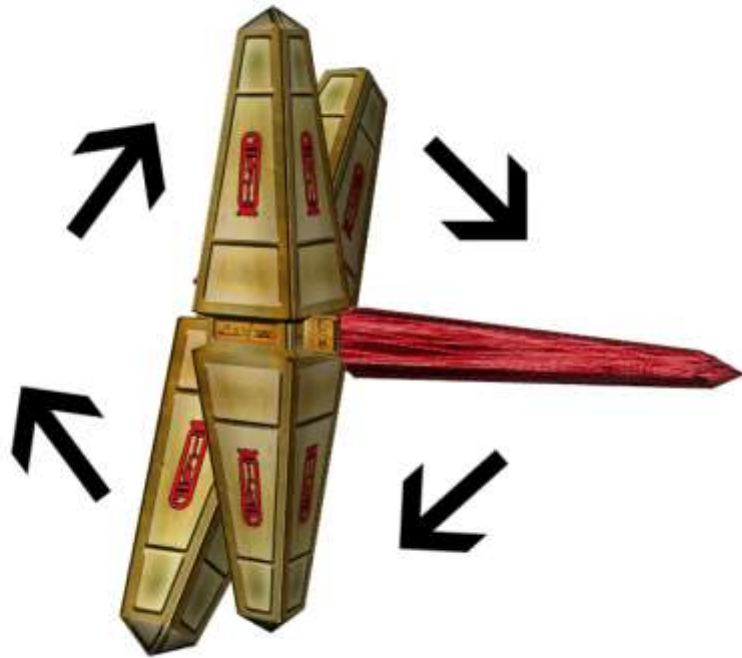
- Fire Gun
- Enabled/Disabled

Attributes

Damage	500
Rate of Fire	0.5 shots/second
Health	∞

Visual Design

Turrets are stationary weapons positioned on the floor. They are one meter high and about one meter wide at the base. The back of the turrets are always spinning. They fire bright blue plasma bolts when triggered.



Name/ID

Console / Object2

Brief Description

The consoles are used to disable turrets in the level. When a console is activated, a turret is turned off so it will no longer attack the player.

The consoles are tall rectangular prisms that are slightly shorter than the player's height and one meter wide. The majority of the console is the same gray and colored metal as the surrounding level environment. The top half of the front of the console is made up of a computer screen and a panel with glowing buttons. When the console has not been activated, the screen and buttons will glow red. After it has been activated, the color of the glow will change to green.

Behaviors

- Perform Action

Attributes

Action	Deactivates a turret.
--------	-----------------------

Visual Design

The consoles are tall rectangular prisms that are slightly shorter than the player's height and one meter wide. The majority of the console is the same gray and colored metal as the surrounding level environment. The top half of the front of the console is made up of a computer screen and a panel with glowing buttons. When the console has not been activated, the screen and buttons will glow red. After it has been activated, the color of the glow will change to green.



Levels and Maps

Name/ID

Mars / 1

Titan / 2

Note: Both levels have the same gameplay.

Brief Descriptions

Mars

The Martian Ziggurat is a combination of close quarters and wide spaces. Though there is a very high ceiling and the upper platforms are all far apart from each other, as well as large flat areas that can be accessed by falling from these platforms, there are also narrow passages on the ground floor. The upper platforms are usually not connected to each other in order to force the player to either jump or run to the ground floor. Turrets guard the flat areas to prevent hiding out in the corners and trying to pick off enemies from a distance. The structures themselves are dark tones of gray with blue highlights, and many sections of walls and floor have glowing patterns.

Titan

This is Ziggurat Six, the most recent addition to the colonial spread of mankind. Titan is one of the farthest-reaching objects in the solar system, therefore, it is even colder and darker than the other Ziggurats. The level interior is very similar to Mars in style and color, but the level itself feels larger; the platforms circle the perimeter and climb progressively higher, leaving the center mostly empty space for Gryphons to patrol. The ramps leading up from the ground floor are symmetrical, and there is only one path to the third floor. Turrets are positioned behind pillars, in corners, or generally in dark areas to surprise the player.

Back Stories

Mars

The Mars Colony was the first ziggurat built off of Earth, therefore, Ziggurat Two is over two hundred years old. It is the first attempt to harness the mystic power for the purpose of fueling surrounding colonies. Though it still functions, it shows signs of its age with cracks, worn surfaces, and very pervasive sand.

This is where the character was stationed as a guard before the monsters attacked.

Titan

Ziggurat Six is the newest attempt at colonization. The Colonial Alliance wants to begin exploring beyond the Kuiper Belt, and investigate reports of strange occurrences happening while in Trans-Neptunian space. Ziggurat Six's construction is incomplete, halted due to mysterious accidental deaths on site. After the recent halt in construction, the site has stood empty for several weeks, during which time the hordes of monsters accumulated before spreading to other ziggurats.

Goal

Your goal is to kill as many enemies as possible (and therefore gain the highest score) before dying, or before the game timer expires. The timer starts at two minutes and you can extend your play time by picking up blue orbs, dropped by enemies.

Level Travel

The player will travel through the level by running, climbing ramps to reach higher platforms, and deactivating turrets to allow further exploration.

Scale

The individual rooms of the level are measured in meters, as detailed on the map. The character is 2 meters tall.

Environmental Interactions - Behaviors

1. Turrets
 - a. These function as hazards in the level. You can deactivate them by finding the corresponding computer console. Otherwise, whenever you get within a predetermined range, they begin attacking you.
2. Consoles
 - a. These allow you to deactivate the turrets mentioned above. They serve no other purpose. Once they have been used they are static and unchanging. When activating a console an alert appears on screen to tell you your action has been carried out successfully.

Ambient Environmental Aspects/Objects in the Level

- **Audio**
 - Music in and out of combat
 - Hum of elevator / Portal / Beam of Light
 - Doors opening and closing
 - Console beep
- **Level Animations**
 - Door opening and closing

Time

Ziggurat Six, by default, has a two minute play timer. You can increase this timer by killing enemies and picking up blue time orbs. Each orb gives you ten seconds more. As time progresses in the game the enemies scale up in difficulty, and every interaction's point reward increases as well. For example, picking up an orb gives you 25 points by default, but after 30 seconds passes, they will give you 50 or more points instead.

Visual Designs

Mars



Titan



Combat System

1. Real-time in-game battles
3. The plasma gun and the pulse cannon damage a single target, while the rocket launcher does splash damage.
4. Player and enemies have a single hit box for damage.
5. When enemies take damage equal to their health, they will die in a cloud of particles. In the model's place will appear floating orbs that will be attracted to the player.

Camera

This image illustrates the distance between the camera and the character.

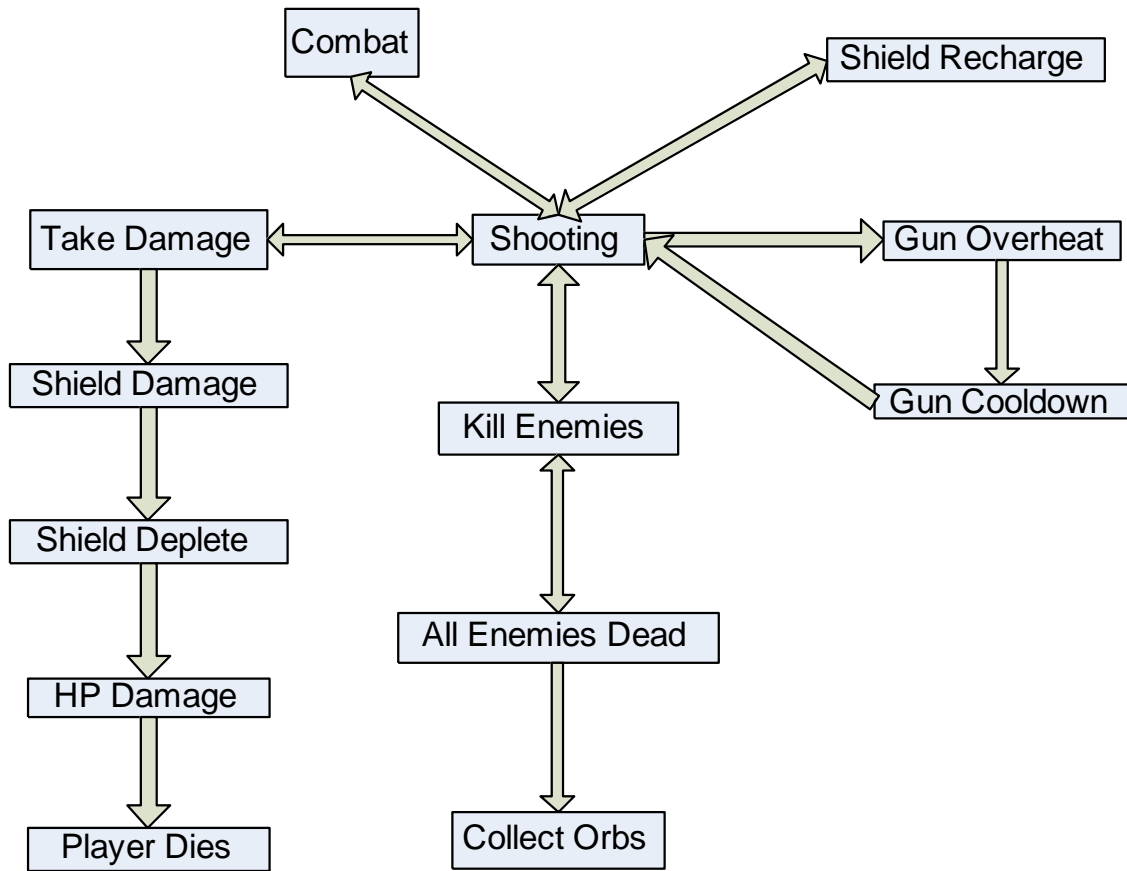


Walkthrough

Normal music will be playing as the player runs into a room full of enemies. Once the player enters combat with the enemies the music will pick up to a much faster pace, and become louder. The ground enemies will be approaching quickly towards your position. The first handful of lasers will be absorbed by your shield. After your shield is depleted your health will start taking damage. At this point you must choose to keep taking damage by staying

in the fight, or to temporarily fall back and let your shield recharge while the enemies chase you. As you are running around the room shooting the enemies, turrets lie in dark places to surprise you with more damage. In addition if you fall off one of the upper platforms you will take damage when you hit the ground. After killing all the enemies and absorbing their orbs, the music will go back to normal.

Flow Chart



Customization Systems

Weapon Upgrading

The weapon being upgraded will be displayed on the bottom right side. The top left side will contain the customization options. These options will contain:

1. Damage dealt
2. Rate of Fire
3. Cooldown rate



Game Logic, Algorithms, and Rules

Key Game Algorithms

AI

The path finding of the enemies in Ziggurat Six consist of a recursive function that cycles through a list of nodes and their paths. While these paths are located, the distance traveled is also saved. After all the paths are found, the shortest one is finally returned and the enemy begins moving towards the next node.

Upgrade System

Each mechanical orb the player acquires is worth one mechanical point. The points are not displayed for the player to see; they are tracked in the game engine. Instead, the player sees a bar on screen slowly filling with color. When the bar fills all the way, the player has gained an upgrade point by acquiring 100 mechanical points. Each orb will be worth 1 mechanical point. The number of upgrade points the player currently has will be displayed in the pause menu.

FAQ

Basics:

When is the game coming out?

The game will be released on October 15, 2007.

What will Ziggurat Six's rating be?

Ziggurat Six will be suitable for teen audiences.

Tech:

What are the system requirements for Ziggurat Six?

1 GHz Pentium III or equivalent, 128 MB of RAM, 64 MB of Video RAM

Will Ziggurat Six take advantage of DirectX 10?

DirectX will not be used. Ziggurat Six is being developed using OpenGL.

Gameplay:

What is the point of the Mechanical Orbs?

Mechanical Orbs are acquired in a similar system to a traditional "Experience Points" value. Basically, each orb is worth a small number of points, and when the player accumulates enough points, they gain an "upgrade point." In the pause menu, it will be possible to spend the upgrade point on a specific key feature. For instance, when you accumulate one upgrade point, you can slightly increase the amount of time you can fire your gun before it overheats and requires a reload.

How do I replenish health?

Health orbs randomly spawn from defeated enemies. When these orbs are collected by the player, a small portion of their health will replenish.

Reference of Key Elements

Scoring

Ziggurat Six does contain a high score system and the main goal of the player is to reach the highest score. As the player defeats enemies, disables turrets, and accomplish other various tasks, he/she will be rewarded points. Another point system in Ziggurat Six is the mechanical/upgrade points. As the player plays the game, they will be able to gain these points in order to upgrade their weapons.

Winning/Losing

When the player first enters a ziggurat, the main goal is to defeat as many enemies as possible and extract orbs from them. There are three different kinds of orbs in Ziggurat Six and each one assists the player in different ways.

Each mechanical orb will give the player a certain amount of mechanical points depending on the difficulty of the enemy that was defeated. When the player gathers a hundred mechanical points, they will be transferred to one upgrade point. Upgrade points can then be used to upgrade weapons in the pause menu at any time. Health orbs replenish the player's health bar and time orbs give you extra time before a game over.

In each level, turrets are scattered to help distract the player while the enemies attack. Turrets do a significant amount of damage but can be disabled at their specific controls panels.

The game ends when either the timer has run out or the player's health has depleted. After the level has finished, the player will be able to enter his/her scores into the high score table if applicable.

Rewards

The orb system is the primary way the player can gain rewards in game. When the player collects enough mechanical orbs, they can upgrade their weapons to be more productive. The mechanical orbs can be spent in the pause menu and upgrades consist of faster rate of fire, more damage, and faster cooldown for your weapons.

When the player finishes the game, they will also be able to enter their name in the high score chart if applicable.

Art and Production Design

Art & Animation Style

The game will take place entirely indoors, inside the ziggurat. The interior of the ziggurats are mostly metal, with lots of glowing lights. Both the Mars and Titan Ziggurats have Egyptian themed hieroglyphs on the walls. The environment is very dark with glowing sections on the enemies, static objects, and level geometry.

Sound Effects Deliverables

All sounds will be a mix between Halo and Star Wars sound effects. Specifically, plasma weapons and laser sounds.

Music Deliverables

There will be 2 forms of music playing through out the game, in-combat and out-of-combat music. The in-combat will be fast paced techno battle music, while out-of-combat will be similar, but slower and softer. Main influences of music will be taken from Metroid Prime and Panzer Dragon Zwei.

Technical Overview

The following pages of this document will function as a guideline for the technical aspects of creation of our game. It is important to us to clearly identify problems early on and create a working game that is as bug-free as possible. We would also like to enter the game into student competitions, and this enhances our desire to make a complete (and fun) game. Our schedules and deadlines are designed around the entrance date for the IGF Student Competition though we hope to use the game in the Full Sail arcade as well – therefore, our technical planning is aiming for a slightly lower amount of memory and technical requirements than simply “it works on the test machine.”

Our coding standards, integration plan, and bug testing have been written only after discussing the options amongst the group. Therefore, we anticipate no problems with any of these important things. For example, we have agreed to all use the Doxygen application to generate HTML tables and diagrams based on our comments; this strictly sets the way comments should be written.

The bulk of the document is in the System Feature Breakdown and the Game Feature Breakdown. The Game Feature Breakdown begins with our Prioritized Feature List that explains what our priorities are as far as gameplay, followed by what modules (both system and game) are required to make it happen.

Coding Standards

Naming Standards

Our naming convention policy is intended to make code more readable and understandable, and therefore more user-friendly. We want such information as functions or macros to be intuitive, so you are never guessing at what the outcome will be or what order to pass in information.

Prefix Convention

We're going to use a format that is similar to Hungarian notation, and capitalization of every word in variables. Class names will not be preceded by a capital C (for instance, `ClassName`, not `CClassName`).

Variables:

```
thing m_MemberVariable; (unique type)
```

```
char *m_szStringThing;
```

```
m_iIntegerExample;
```

Functions:

```
int PublicFunctionThing(parameter, whatever);
```

```
int PrivateFunctionThing(parameter, whatever);
```

This is the same format for all functions regardless of their accessibility (public, private, protected).

Classes

For explanations of commenting, see the Commenting section on page 8.
Example of naming and formatting:

```
/** The derived classes are listed on the same line as the class
 * declaration.
 */
class ClassName : public ParentClass
{
public:
    /** Explain the variable here. */
    int whatever; /**< Or explain the variable here instead */

    /** Explain the function
     * @param     ParameterName     Comment about the parameters here
     * @return     Comment about return value here
     * @sa     Class name, function name, etc. (See Also)
     */
    int PublicFunctionThing(type ParameterName);

private:
    /** Lengthy explanation of the following variable. Use this
     * format if your explanation is going to be longer than a few
     * words.
     */
    type m_MemberVariable;
    char *m_szStringThing; /**< Short explanation here. */

    /** Use same format as previous function */
    int PrivateFunctionThing(parameter);

protected:
    type *m_pPointerExample; /**< Short explanation here. */
    type *m_iIntegerExample; /**< Short explanation here. */

};
```

Example of code inside a .cpp file:

```
int ClassName::PublicFunctionThing(type ParameterName)
{
    /** Explain what your function is about to do. Try not to have
     * more lines of commenting than lines of code. If your function
     * body is less than 5 lines then it might happen anyway.
     */
    Code_here();
}
```

Structures

If you have to add more than default constructors, make it a class!

Variable and structure names follow the same naming conventions as classes.

```
struct Example
{
    thing m_MemberVariable; /**< Short explanation here. */
};
```

Relevant Function Names

Our functions will not be prefixed by any notation. It will just be:

returnvalue FunctionName(parameter);

Function names will not be longer than 4-5 words maximum (excluding words like “from” or “a” which are obviously needed at times).

Macros and Constants

Constants

Preprocessor: Only needed for hard-coded number such as maximum sizes of various things (Player name, etc), or default values to help centralize data.

This is the only kind of constant we are going to implement. It will be a #define format that will be declared in the respective header file of whatever it is primarily related to. For example, a #define for the default window size would belong in a rendering header file.

Macros

Safe release of pointers – Check if pointer is valid, if so, set to null, and so forth.

If we need to implement bit manipulation it will be done via macros as well.

All will be declared in their respective header files. Constants will be in all caps, with words separated by underscores (i.e. PLAYER_NAME). Macros will be named the same way as functions with similar rules to parameter names, commenting, etc.

Summary of Naming Convention

Variable names will be prefixed by an m_ if it is a member variable. Capitalize each word in a variable, class or function name. Constants should be in all caps with underscores separating individual words.

Commenting

We will be implementing Doxygen, a program that allows generation of html charts and documents based on comments imbedded in our code files. Therefore, we will be following specific code standards in that regard.

The beginning of a Doxygen comment needs to open with `/**` in order to work. Also, any Doxygen keywords are prefixed by the `@` symbol.

Examples:

```
/**
 * @brief Brief description of whatever this comment block relates to.
 * In this case it will be tied to the following class declaration.
 * This should only be one sentence long or it will be separated into
 * an Extended Description field.
 *
 * @file The file path of this code. It can be as simple as FileName.h
 * @author Viking Jack Development Team
 */

class ClassName : public ParentClass
{
public:
    /** Explain the variable */
    int whatever; /**< Or explain here instead; < symbol required */

    /** Explain the function
     * Brief description of what this function does. Anything beyond
     * one sentence will be separated by Doxygen into an Extended
     * Description field, so make sure the first sentence is
     * fundamental.
     *
     * @param      FunctionParameter      Comment about the parameters
     * @return      Comment about return value here
     * @sa      (See Also)Class name, function name, etc. List
     *           names of anything that is related to this function.
     */
    int PublicFunctionThing(FunctionParameter);
};
```

The difference between `.h` and `.cpp` commenting will be related to function explanations. The header files will contain detailed comment blocks preceding each function. The `.cpp` will not have these comment blocks, but commenting is expected throughout function code. The information necessary for these blocks is, at minimum, explanation of every parameter and return type (if any), and a two-sentence description of the function's overall purpose. Commenting inside

function bodies will be at minimum a one sentence explanation every five lines of code (there are exceptions for extremely short function bodies – for example one that is only ten lines or so). However, there should not be more commenting than code. If you are performing an operation combined into one line that requires a paragraph of explanation, you should most likely break the operation into steps (such as ternary operators inside other ternary operators and so on).

Data Alignment

Data will be 4-byte aligned. We will enforce this only lightly because we are not concerned about running out of RAM or even approaching the maximum limit. However, it is suggested that members follow 4-byte alignment when making structures.

Coding Guidelines

1. Anything that is memory managed needs to be derived from the Object class.
2. Singletons must publicly derive from a Singleton class.
3. The constructor of every class will initialize all data members.
4. Function prototypes do not require default arguments.
5. Inline functions are not allowed to call functions. Macros should be used sparingly inside these functions; only when necessary.
6. Do not use global variables. One alternative is a static member variable.

Lua Exposure

There are a few things that need to be done in order to allow your module to be accessible from within the Lua engine.

What goes in your class declaration:

```
LUA_METHODMAP();
```

This macro injects the Lua method table into the class. Semicolon optional.

```
static void RegisterLuaInterface(lua_State *L);
```

This function is used to register all the methods bound to Lua in the class.

```
static int lb_[methodname](lua_State *L);
```

For each method you're exposing to Lua, define a function according to the prototype above. For example lets say you want to map `SomeClass::Bounce()` to `someClass.bounce()` in Lua. The correct function name would be `lb_Bounce`. This is not strictly enforced and is just a naming convention.

What goes in your class implementation:

```
BEGIN_LUA_METHODMAP(ClassName)
```

This macro is used similarly to MFC's message map macros. This is placed at the top of the class's source file, passing in the class's name where `ClassName` is. Note that there is **no** trailing semicolon!

```
END_LUA_METHODMAP()
```

This macro ends the method map started with the `BEGIN_LUA_METHODMAP()` macro. It follows all the `LUA_METHOD()` calls. Trailing semicolon is optional.

```
LUA_METHOD("luaname", lb_[methodname])
```

For each `lb_method` you defined in the class declaration, a correlating call to this macro is required. The first parameter is a string of what the method name will be inside Lua and the second parameter is the function name defined in the header. Once again, note the lack of a trailing semicolon!

Additionally, don't forget to implement the `lb_method` functions in the class implementation.

Basic Lua C API usage

Lua operates entirely on a stack-based system, this is not unlike that of assembler. Arguments are provided from Lua to C using values pushed onto the stack. Likewise, return values from C are given to Lua by pushing them onto the stack.

How to get arguments from Lua

```
luaL_check[(L, stack position)]
```

The luaL functions are used to both check if the item at the specified stack position is the correct type, as well as returning the item from the stack. Now how the hell do we find out where an argument is on the stack? Negative numbers! Any negative number denotes its position, starting from the top which is referred to as -1. -2 is the next one and so on. Arguments from Lua are pushed on in the order that they are passed in to the function. So therefore:

```
somefunction(arg1, "arg2")
```

would translate to C as:

```
int arg1 = luaL_checkinteger(L, -2);  
const char *arg2 = luaL_checkstring(L, -1);
```

Note that this is backwards, and the numbers change depending upon how many arguments are passed in. To determine how many arguments that are being passed in, lua_gettop(L) can be used. The following luaL functions pair up to the corresponding C datatypes:

```
luaL_checkstring = const char *  
luaL_checkinteger = int  
luaL_checknumber = float  
luaL_checkboolean = bool
```

There are a few more functions, but they probably aren't necessary. Refer to the Lua documentation for more info.

Returning values to Lua

Since Lua does operate on a stack, values must be returned on the stack! The way we do this is push the respective values onto the stack, and then return the

number of values we pushed (this is why our C functions always return an int). The following methods can be used to push C datatypes to Lua:

```
lua_pushboolean    = boolean
lua_pushinteger    = int
lua_pushnil        = null-like
lua_pushnumber     = float
lua_pushstring     = char *
lua_pushlightuserdata = void *
```

Pretty logical, it's just the reverse of `luaL_check[]` (and `lua_to[]`). One thing to note is `lua_pushlightuserdata`. This is a pretty odd function as it doesn't correlate to any real Lua data type, since they don't have pointers. How light user data would be used is to have 'objects' in Lua that link back to actual objects in C++. A way to implement this is by pushing and popping the this pointer of a C++ object.

For more info and a great reference on Lua check out:

<http://www.lua.org/manual/5.1/>

Development Environment

This project will be using C++, written via Visual Studio 2003, as well as Lua 5.1 to create our core game functionality. We are also going to be using additional tools to help productivity, such as Doxygen to create dynamic documentation, as well as 3D creation via Maya 8.5.

Technical Creation Tools

- Visual Studio 2003
- Lua 5.1
- OpenAL 1.1
- OpenGL 2.0
- RenderMonkey 1.6.2
- Simple DirectMedia Layer 1.2.11

Asset Creation Tools

- Maya 8.5
- Photoshop CS2

Miscellaneous Tools

4. Doxygen 1.4.6

Timing Specifications

We are aiming for 50 fps. During each frame the breakdown will be as follows:

Rendering: 30%

Collision: 20%

Input updates: 5%

AI: 30%

Lua overhead: 10%

Sound: 5%

System Feature Breakdown

Module Breakdowns

Rendering Engine

The rendering engine provides an abstract, modular interface to a rendering system that does not rely on api-specific features. The system is developed primarily with OpenGL but a Direct X port requires little to no change with the given interface. Included in the rendering system is an optimized scene graph system which uses a variety of sorting and retained information to accelerate rendering speed.

Dependencies

- Modules that depend on it:
 - Model loader, animation system, texture manager, camera, character manager, enemy manager, weapon manager, boss manager, orb manager, level manager

Features

- Scene Graph/Manager
 - State sorting (shaders, materials, textures).
 - Optimized culling using retained information.
- Shader support
- Point sprite support
- Framebuffer support
- Vertex Buffer Object support

Example: Loading and rendering a mesh (w/o scene management)

```
// Create and initialize the renderer
RenderDevice *renderer;
renderer = (RenderDevice *)CreateRenderDevice(RT_OPENGL);
renderer->SetSwapFunc(SDL_GL_SwapBuffers);
renderer->DepthTest(true, CF_LESS);
renderer->GoPerspective(45.0f, 0.1f, 500.0f);
renderer->SetClearColor(Vector4(0.3f, 0.3f, 0.3f, 0.0f));
```

```
// Load the mesh
Mesh *m = renderer->LoadMeshFromFile("droid.md2");
```

```
// Render
while (true)
{
    renderer->Clear();
```

```
m->Draw(renderer);  
renderer->Swap();  
}
```

Time to Complete Estimate

- Initial coding setup: 5 days
- Testing: 4 days
- Scene Manager research: 1 day
- Scene Manager implementation: 3 days
- Scene Manager testing/optimizing: 4 days
- Vertex Buffer Object support: 2 days

Module Author(s)

- Josh Stefanski

Scene Graph

The scene graph module organizes the renderable scene into a logical hierarchy. This hierarchy is then traversed and organized into material-specific lists to optimize rendering and the associated state changes. Scene nodes provide a means of positioning objects in the scene as well.

Dependencies

- Depends on the following modules:
 - Rendering Engine

Features

- Hierarchical organization of SceneNodes
- Provides multiple render passes based on material type
- Optimizes state changes and buffer writing

Time to Complete Estimate

- Research: 2 day
- Base classes: 1 day
- Additional nodes: 2 days
- Optimization: 1 day
- Testing: 4 days

Module Author(s)

Josh Stefanski, Joe Leigh

Model Loader

This module provides a way to load COLLADA asset files into a Mesh object. Models must be exported with the latest version of ColladaMaya, Feeling Software's COLLADA exporter. FCollada, their import and export library will be used for low-level parsing of loaded models.

Dependencies

- Depends on the following modules:
 - Rendering Engine
- Modules that depend on it:
 - Animation System

Features

- Loads DAE files
- Keeps track of models loaded
- Prevents redundant model loading

Time to Complete Estimate

- Researching FCollada library: 1 day
- Loading and rendering static meshes: 1 day
- Loading and displaying joint information: 2 days
- Testing: 4 days

Module Author(s)

Josh Stefanski

Model Manager

The model manager provides a cache-enabled interface to loading models from a file. This prevents duplicates of the mesh in memory.

Dependencies

- Depends on the following modules:
 - Rendering Engine
- Modules that depend on it:
 - Animation System

Features

- Prevents redundant model loading

Time to Complete Estimate

- Interfacing with model loader: 1 day

Module Author(s)

Josh Stefanski

Texture Manager

The texture manager provides a means of keeping track of already loaded textures, therefore preventing duplicate copies in memory. It interfaces indirectly with the rendering system using predefined types.

Dependencies

- Depends on the following modules:
 - Rendering engine
- Modules that depend on it:
 - HUD, menu manager, level manager, character manager, boss manager, enemy manager, weapon manager, model loader

Features

- Transparent interface for texture loading
- Keeps track of already loaded textures so the game doesn't need to.

Basic Example

An enemy texture is loaded during the first floor of the first level. Then during a floor or level change a texture is requested again, but since it is already loaded in memory, it just returns the existing object.

Associated Risks

5. Insert any risks that we have determined to be associated. The risks should be in the table format.

Time to Complete Estimate

6. Creation: 3 days
7. Testing: 1 day

Module Author(s)

Joe Leigh

Animation System

This module will animate models using skinning. Both software and hardware skinning will be available, hardware using GLSL. This module is deeply integrated with the model loader.

Dependencies

- Depends on the following modules:
 - Model Manager, Renderer

Features

- Loads DAE files
- Keeps track of animations

Time to Complete Estimate

- Researching skinned animation: 1 day
- Rendering and animating joints: 3 days
- Deforming mesh according to joints: 3 days
- Testing: 4 days

Module Author(s)

Josh Stefanski

Collision

This module takes care of all the collision detection in the game. The entire room of the level will be put into an octree. The collision module will iterate through the tree, checking each node if it has triangles in it. When the current node being checked is a leaf node, contains triangles, and is being collided with, the module will check each triangle within the node for collision. If a collision has occurred, the module will then call the necessary response functions depending on the situation.

Dependencies

- Depends on the following modules:
 - Math library
- Modules that depend on it:
 - Character manager, enemy manager, orb manager, enemy manager

Features

- Contains functions needed to detect collision detection and call the correct response functions.
- Octree
 - Maintains the octree and all of its children. This will be used to break down the scene for faster collision detection.

Examples

```
// Our point to AABB collision test
bool PointToAABBCollision(Vector3 point, AABB aabb) {
    return ( point.x > aabb.GetMin().x && point.x < aabb.GetMax().x &&
            point.y > aabb.GetMin().y && point.y < aabb.GetMax().y &&
            point.z > aabb.GetMin().z && point.z < aabb.GetMax().z );
}
```

Time to Complete Estimate

- Octree creation: 3 days
- Collision task creation: 4 days
- Collision on entities: 2 days
 - Complete
- Testing: 3 days

Module Author(s)

Steve Chester

Camera

The camera module consists of two parts: A camera scene node implemented within the rendering system's scene manager and a lighter-weight class used to manipulate the camera with respect to the player.

Dependencies

- Depends on the following modules:
 - Rendering system, character manager

Features

- Camera Scene Node:
 - Provides common camera transforms
 - Contains a frustum for culling purposes
- Camera interface class:
 - Links to the camera scene node
 - Provides a script interface to logical transforms
 - Attaches to the player class

Input-controlled Camera Example

Utilizing the scripting interface, the following could be used to control the camera's movement within a script:

```
if (input.keyDown(KEY_W)) then
    camera.moveForward(PPLAYER_MOVE_SPEED)
end
/* etc... */
if (input.mouseMovedRelative()) then
    camera.rotateY(input.mouseDX() * MOUSE_SENSITIVITY)
    camera.rotateX(input.mouseDY() * MOUSE_SENSITIVITY)
end
```

Associated Risks

- Insert any risks that we have determined to be associated. The risks should be in the table format.

Time to Complete Estimate

- Creation: 3 days
 - Complete

Module Author(s)

Joe Leigh

Input

This module enables detecting a variety of input types from both the keyboard and mouse. Key-binding is included as well. It is also responsible for controlling the input received from the Xbox360 controller. It will be using XInput and be a secondary control option, over the mouse and keyboard. All keys will be mapped from the keyboard to make for natural play using the controller.

Dependencies

- Modules that depend on it:
 - Character manager, weapon manager, level manager, menu manager

Features

8. Immediate input
9. Buffered input
10. Timed/Delayed input
11. Relative and absolute mouse input

Controls

12. Movement
 - 13. Forward/Backward = W or S on keyboard, left joystick on controller
 - 14. Look/Turn = Mouse, right joystick
 - 15. Strafe = A and D on keyboard, left joystick on controller
 - 16. Fire weapon = Left click on mouse, right trigger on controller
 - 17. Switch weapon = Tab on keyboard, left or right bumper on controller
 - 18. Lock on to enemy = Right mouse button, left trigger on controller
 - 19. Activate object = E on keyboard, A on controller
 - 20. Enter pause menu = Escape on keyboard, start on controller
 - 21. Navigate menu = Mouse, left joystick on controller

Time to Complete Estimate

22. Keyboard/Mouse wrapper creation: 1 day
23. Complete
24. Xbox 360 wrapper creation: 2 days
25. Testing: 1 day

Module Author(s)

Josh Stefanski, Joe Leigh

Sound Manager

This module will manage sound and music. It will allow control of the music playback, as well as smoothly transition between songs. It also prevents the same sound from being loaded again if it's currently in memory.

Features

- Loads sounds and music
- Streaming sound
- Smoothly transition from exploring music to battle music
- Playback control

Time to Complete Estimate

- Loading: 1 day
- Streaming: 1 day
- Transitions: 1 day
- Controlling Playback: 1 day
- Testing: 3 days
- Integration: 1 day

Module Author(s)

Ernesto Rojo Jr

Particle System

This module is responsible for creating and handling all particle effects in the game. It will use OpenGL point sprites to render its effects. It will also create and delete the emitters used in the game to show the portals. All particle effects will be scripted and have the option to be billboarded towards the camera.

Dependencies

- 26. Depends on the following modules:
- 27. Texture manager, timer

Features

- 28. Emitter Object
- 29. Will load a pre-made particle effect at a specified position.

Emitter Script Example

```
30. EMMITER teleporter{  
        Effect(particle_fire),  
        Position(0.0, 1.0, 0.0)  
};
```

Time to Complete Estimate

- 31. Interpolation: 1 day
- 32. Editor conversion to OpenGL: 1 day
- 33. Point sprites: 2 days
- 34. Testing: 1 day
- 35. Creation of particle effects: 2 days
- 36. Integration: 1 day

Module Author(s)

Joe Leigh

HUD

The HUD is responsible for drawing and updating HUD elements as well as the aiming reticle. Such elements include: the player's health, the amount of upgrade points available, shield strength, and weapon's temperature.

Dependencies

- 37. Depends on the following modules:
- 38. Upgrade manager, Level manager

Features

- 39. Centralized drawing of HUD elements
- 40. Keeps track of HUD values locally
- 41. Displays health bar, mechanical orbs, and current weapon temperature
- 42. Displays currently equipped weapon

Player Life Change Example

Precursor: The player has taken damage and holds the new data. The player will then update the HUD by sending its new health count. The next time the HUD draws, it will update accurately.

Time to Complete Estimate

- 43. Player information: 4 days
- 44. Integration: 1 day
- 45. Asset implementation: 3 days

Module Author(s)

Ernesto Rojo Jr.

Memory Manager

The memory manager is a wrapper for logging calls to new and delete which will allow tracking of memory chunks to find leaks and unnecessary deallocations and crashes. The originating call will have its filename and line number saved.

Dependencies

- No hard dependencies

Features

- (De)Allocation logging.
- Debug output for mitigating memory leaks.

Time to Complete Estimate

- Memory manager creation: 1 day
- Logging creation: 1 day
- Logging implementation: 1 day
 - Complete

Module Author(s)

Josh Stefanski

Scripting

The scripting module encompasses two different areas: The scripting base interface and implementation as well as each specific module's scripted interface. The base interface deals with scheduling script execution/caching and a macro-backed implementation for allowing other modules to provide an interface inside Lua. Each module's scripted interface will provide function calls and objects for manipulating the respective module components within the game scripts for a truly data-driven design.

Dependencies

The following modules depend on the basic Lua implementation, however the scripting system as a whole depends on these modules as well for their scripting interface.

- Modules that depend on it:
 - Character manager, sound manager, weapon manager, enemy manager, orb manager, boss manager, level manager, menu manager, camera, input, HUD, portals

Features

- Lua-based scripting system
- Engine and game objects exposed to Lua
- Script scheduling for periodic execution
- Script caching for optimization

Time to Complete Estimate

- Interface/general setup: 2 days
- 1 day for each module that uses scripting functionality

Module Author(s)

Josh Stefanski (Base interface)

Kernel

The kernel is a object-oriented alternative to a main game loop. The kernel comprises of a list of tasks and what priority or 'slot' they are placed in to determine the order of execution. Suspending and resuming of tasks is easy and allows switching between groups of tasks to be done with relative ease. Each task has a callback for being started, stopped, and updated. Additionally, handlers for when the task is suspended or resumed are provided. The kernel sorts tasks based on their respective priority when added, with lower numbers being executed first.

Dependencies

- Modules that depend on it:
 - Model manager, texture manager, animation system, collision, input, sound manager, particle system, HUD, memory manager, logging, scripting

Features

- Object-oriented approach to a game loop
- Suspending/Resuming support

Time to Complete Estimate

- Interface/general setup: 2 days
- 1 day for each module that uses scripting functionality

Module Author(s)

Josh Stefanski (Base interface)

GlobalTimer

This module provides overall timing for the engine. Average FPS and associated frame time statistics are calculated. Additionally, a smaller timer object interface exists to be used in alarm-clock situations within the code and the scripts.

Dependencies

- Modules that depend on it:
 - Animation system, collision, input, sound manager, particle system, HUD, memory manager, scripting

Features

- Overall timer statistics (frame time, average frame time, fps)
- Timer object support

Time to Complete Estimate

- Interface/general setup: 1 day
- Script interface: 1 day

Module Author(s)

Josh Stefanski

Shaders

This module is composed of two GLSL shaders: Basic lighting, Light mapping. Basic lighting operates on a virtual lighting implementation. The vertex shader calculates the distance and attenuation to the three closest lights, while the fragment shader calculates color based on those same three lights. Light mapping enables the use of 'fake' lighting techniques implemented with multitexturing.

Dependencies

- Modules that depend on it:
 - Animation

Features

- Basic per pixel lighting
- Light mapping

Time to Complete Estimate

- Interface/general setup: 2 days
- Basic lighting: 4 days
- Light mapping: 2 days

Module Author(s)

Joe Leigh, Ernesto Rojo Jr.

Game Feature Breakdown

Prioritized Feature List

A Features

- Movement around a level
 - Technology Required: Loading textured model; Animation of running; Collision between player, walls and floor; Input
 - System Modules: Model loader, animation system, collision, input wrapper, camera class
 - Game Modules: Character class, level manager
- Single type of enemy – walking
 - Technology Required: Animation of running; Collision between the enemy and player, walls, and floor
 - System Modules: Model loader, animation system, collision
 - Game Modules: Enemy class, weapon manager
- Ranged combat between player and an enemy
 - Technology Required: Animation of firing weapons for player and enemy; Collision between projectiles and enemy/projectiles and player; Sound manager
 - System Modules: Collision, input wrapper, sound manager
 - Game Modules: Weapon manager, player class, enemy class
- Mars level
 - Technology Required: Camera system; Loading models; Level manager
 - System Modules: Model loader, camera, collision, particle system
 - Game Modules: Level manager
- Switch between two types of ranged weapons
 - Technology Required: Multiple weapon models; Weapon base class and two derived classes
 - System Modules: Model loader, collision, input
 - Game Modules: Weapon manager, level manager
- Locked doors, opened by interaction with stationary objects
 - System Modules: Model loader, input, animation
 - Game Modules: Level manager
- Keyboard/Mouse Input
 - Technology Required: Input wrapper

- System Modules: Input wrapper
- Game Modules: None
- Health Orb system
 - Technology Required: Additional asset, managing class
 - System Modules: Model loader, collision, enemy class
 - Game Modules: Orb manager

B Features

4. Third type of ranged weapon – fire
 5. Technology Required: Additional weapon model; Particle effect; Third derived class
 6. System Modules: Model loader, collision, input
 7. Game Modules: Weapon manager
8. Second type of enemy – flying
 9. Technology Required: Animation of flying; Collision between the enemy and player, walls, floor; Additional enemy class for AI & attacks
 10. System Modules: Model loader, animation system, collision
 11. Game Modules: Enemy class, weapon manager
12. Weapon Orb system
 13. Technology Required: Additional menu for upgrades; Mechanical points implemented & drops from enemies
 14. System Modules: Model loader, collision
 15. Game Modules: Orb manager, weapon upgrade system
16. Stationary turrets functioning as hazards
 17. Technology Required: Additional class for AI
 18. System Modules: Model loader, collision
 19. Game Modules: Enemy class, weapon manager

Module Breakdowns

Weapon Manager

This module is in charge of all weapons used by every entity in the game. This includes not only the character, but also the enemies and boss of the game. Each entity contains a weapon pointer representing the currently equipped weapon. This determines how much damage the entity does, rate of fire, special conditions, etc.

Dependencies

- Must be completed to reach 100% completion:
 - Rendering engine, model loader, input, orb system, texture manager
- Depend on this module to be completed:
 - Character class, enemy manager, boss manager, orb system

Algorithms

- Ray casting collision to determine if the gun hits a target.
- If the collision is true, the target loses the determined amount of health.
- When the rocket launcher collides with any object, including a wall, it will calculate a sphere with the center point of the collision point. Then it will check any entities inside the radius with a sphere to AABB collision check.

Additional Information/Examples

- There are four types of weapons in the game. They all function like guns in that they have a rate of fire and damage per hit. Every shot adds to a total heat value that is capped at 100 for each weapon. When the gun reaches 100 heat it becomes unusable and will need to cool down before it can be shot again. When the cool down time is reached it will lose heat at a specific rate of heat per second, unique to each weapon.
 - Plasma pistol
 - Rate of fire: 2 shots/second
 - Damage: 50
 - Heat per bullet: 2
 - Cool down time: 2 sec
 - Cool down rate: 10/sec
 - Plasma machine gun
 - Rate of fire: 6 shots/second
 - Damage: 20

- Heat per bullet: 1
- Cool down time: 2 sec
- Cool down rate: 10/sec
- Rocket Launcher
 - This weapon shoots like a rocket launcher or a grenade in that it has a blast radius on impact. Otherwise it is identical to the other weapons.
 - Rate of fire: 0.5 shots/second
 - Damage: 100
 - Heat per bullet: 100
 - Cool down time: 2 sec
 - Cool down rate: 10/sec
 - Blast radius: 1 meter
- Lightning spell
 - Rate of fire: 0.5 shots/second
 - Heat per bullet: 100
 - Cool down time: 2 sec
 - Cool down rate: 10/sec

Time to Completion

- Projectile creation: 1 day
- Plasma pistol: 1 day
- Plasma machine gun: 1 day
- Weapon switching: 1 day
- Collision testing: 2 day
- Integration: 1 day
- Weapon overheats: 1 day
- Balance testing: 1 day
- Rocket launcher: 2 days
- Balance testing: 2 days
- Upgrade points: 4 days

Module Authors:

Joe Leigh, Jamie Cannon

AI Manager

The AI manager encompasses all the AI-controlled actions and entities within the game such as Enemies and Turrets.

There are two fundamental types of enemies in our game: a walking enemy and a flying enemy. Both function in the same basic way with a single ranged attack. However, the flying enemy moves closer to the player whereas the walking enemy stays further away while attacking.

There is a third more simple type of enemy as well: the Turrets. A Turret is stationary and fires forward of its current orientation. It fires constantly while the player is in range unless it is deactivated. Deactivation occurs when the player interacts with a specific computer console.

Dependencies

- Must be completed to reach 100% completion:
 - Rendering engine, model loader, collision, weapon manager, texture manager
- Depend on this module to be completed:
 - Orb system, HUD

Algorithms

- Walking Enemy
 - Determines if it is within 4 meters of the player
 - If so, moves back to be more than 4 meters away
 - Calculates a random percentage to determine if it successfully hits the player (50% chance of success)
- Flying Enemy
 - Determines if it is more than 2 meters away from the player
 - If so, moves to within 2 meters of the player's position
 - Attacks with breath attack
- Turret
 - Checks if there is a player within its range of attack
 - Checks if its current state is active or not
 - If it is active begins firing regardless of exterior conditions, until the character moves out of range or is dead.

Additional Information/Examples

- The enemies all have the same basic attributes as weapons. The flying enemy's attack is a cone of breath so it also has two restrictions: range and angle of fire.
- Walking Enemy
 - Health: 500
 - Movement speed: 2 m/sec

- Rate of fire: 2 shots/sec
- Damage: 15
- Flying Enemy
 - Health: 400
 - Movement Speed: 3 m/sec
 - Range: 2 m
 - Angle of fire: 15 degrees
 - Damage per second: 6
- Turret
 - Health: N/A
 - Rate of Fire: 1 shot/sec
 - Damage: 500
 - Range: Determined by case-by-case basis.

Time to Completion

- Enemy 1 creation, including weapon: 2 days
- Enemy 1 movement and shooting AI, including integration and testing: 8 days
- Enemy 2 creation, including weapon: 2 days
- Enemy 2 movement and shooting AI, including integration and testing: 8 days
- Turret functionality: 1 day
- Turret integration and testing: 2 days

Module Authors:

Steve Chester, Jamie Cannon

Projectile Manager

All projectiles will reside locally within the projectile manager for ease of bookkeeping. The manager will be responsible with spawning, updating, and destroying projectiles of all types.

Dependencies

- Mutual dependencies:
 - Weapon manager, HUD

Time to Completion

- Base setup: 1 day
- Testing: 1 day

Module Authors:

Joe Leigh

Orb Manager

There are three types of orbs: health orbs, time orbs, and mechanical orbs. They are dropped in random amounts by enemies upon death. They move straight toward the player upon creation.

Dependencies

- Must be completed to reach 100% completion:
 - Rendering engine, model loader, collision, character manager, enemy manager, weapon manager
- Depend on this module to be completed:
 - Weapon manager, HUD

Algorithms

- Orbs will move toward the player slowly (0.5 m/s) from the instant they are created.
- There is no pathfinding on the orbs; they will move through the walls.
- Health orbs will drop at a rate of $\text{rand}() \% 4 + 5$.
- Mechanical orbs will drop at a rate of $\text{rand}() \% 3 + 1$.
- Collision will be determined by a sphere to AABB check.

Additional Information/Examples

- Orbs have one general value, that being movement speed.
- Health Orbs
 - Health restored: 20
 - Movement speed: 0.5 m/s
- Mechanical Orbs
 - Points per orb: 1
 - Movement speed: 0.5 m/s

Time to Completion

- Health orbs created: 1 day
- Collision testing: 1 day
- Movement: 1 day
- Mechanical orbs created: 1 day
- Shader application: 1 day
- Testing: 1 day
- Integration: 2 days

Module Authors:

Joe Leigh

Level Manager

The level manager will manage the current floor and transition between the floors directly connected to it. It will also handle user interaction with static meshes that trigger events, such as turret disabling and opening doors.

Dependencies

- Must be completed to reach 100% completion:
 - Rendering engine, model manager, collision, portals, texture manager
- Depend on this module to be completed:
 - Portals

Algorithms

- Door Trigger
 - if (distanceToPlayer < 1M) openDoor();
- Turret Trigger
 - if (distanceToPlayer < 1M) disableTurret();

Additional Information/Examples

- A single level consists of several floors. A floor is defined as the space between two portals, consisting of rooms and hallways. Rooms normally contain enemies, traps and static meshes. Floors are self-contained, like the floor of a building and are connected only by the portals.

Time to Completion

- Loading level information, including position of trigger points: 1 day
- Interaction with objects: 1 day
- Transition to next floor: 1 day
- Add Mars 2F, 3F, 4F: 3 days

Module Authors:

Joe Leigh

Menu Manager

This module will handle the rendering and handling of menus. It will work on a stack system, having the current menu on top of the stack. When a user backs up from the current menu, the top of the stack is popped off and the next item is rendered.

Dependencies

- Must be completed to reach 100% completion:
 - Texture manager, input

Additional Information/Examples

- Example of stack contents
 - Sound Options
 - Options
 - Main Menu

Time to Completion

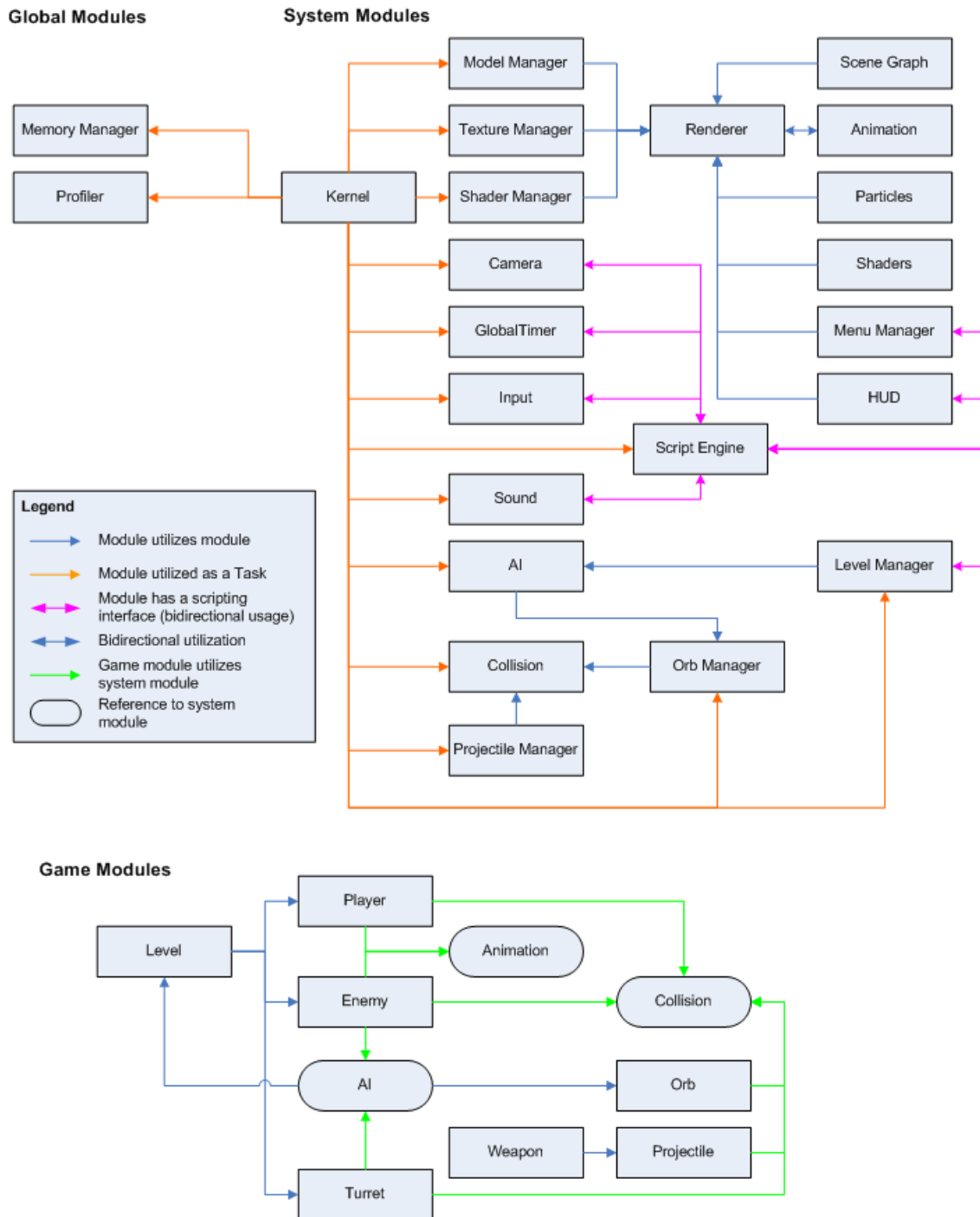
- Options: 1 day
- Pause, Game Over, Intro: 1 day
- Testing: 1 day
- Upgrade system: 1 day
- Character customization: 1 day
- Testing: 1 day
- Integration: 1 day

Module Authors:

Ernesto Rojo Jr

System Architecture

The system is designed to provide a script-driven engine to interact with all the technical sub-systems required to run the game. Scripts will have the ability to interact with the rendering, input, sound, and GUI subsystems through exposed objects. All dynamically allocated memory is tracked by the memory manager to allow for garbage collection and logging. The texture, model, and animation managers provide a layer of abstraction in order to avoid duplicate tasks (loading a texture or model twice, etc).



The diagram above shows the top-level interactions between key subsystems within the engine and game. The blue arrows are used to show when a module utilizes functionality from another module. Orange arrows are a slight variation on the blue arrows where the module isn't directly known to the originating module, in all cases this represents the Kernel/Task relationship. The kernel class itself does not know of all the modules that will be injected into the kernel at runtime, but only certain modules are utilized as a task. Blue double-

ended arrows are exactly like the blue arrows except there's a mutual dependency between the linked modules. Pink double-ended arrows are one of the most prominent relationships, depicting where a module has part of it exposed to scripts through the scripting engine. The reason why the arrows are doubled-ended is to show bidirectional data transfer; Scripts can send information to that module and receive information from the module. Green arrows originate from game modules only and can only point to system module references (these are the rounded rectangles). These arrows show that the specified game modules interact with the respectful system modules. The main reason for this additional arrow type is for a cleaner diagram.

Module-specific relationships

Utilization Relationships

Model Manager → Renderer

The model manager loads and stores models created through the rendering engine.

Texture Manager → Renderer

The texture manager uses the rendering engine for the loading and creation of texture objects, but stores them locally for quick management.

Shader Manager → Renderer

The shader manager uses the rendering engine to load and create shader objects but stores them locally for quick caching.

Scene Graph → Renderer

The scene graph invokes renderstate changes on the rendering engine when drawing the scene.

Particles → Renderer

Particles are managed entirely with the particle system but drawn through the point sprite interface the rendering engine provides.

Shaders → Renderer

The GLSL shaders are loaded in and applied through the render engine's shader interface.

Menu Manager → Renderer

The renderer's 2D drawing interface is used for displaying the menus.

HUD → Renderer

The renderer's 2D drawing interface is used for displaying the HUD elements.

Level Manager → AI

The level manager creates and manages entities within the level, but passes them to the AI module for processing.

AI → Orb Manager

The AI module invokes orb spawning through the orb manager.

Orb Manager → Collision

The orb manager updates all active orbs and performs collision checks through the collision system.

Projectile Manager → Collision

The projectile manager spawns projectiles which are then added to the collision system.

AI → Level

The AI is aware of what exists within the level, so appropriate accessors are utilized.

Level → Turret

Levels hold a list of turrets that are active.

Level → Enemy

Levels hold a list of enemies that are active.

Level → Player

The level manages updating the player and keeping track of the player's status in the game.

AI → Orb

The AI spawns orbs when the appropriate entity is destroyed.

Weapon → Projectile

Weapons fire projectiles and use this module to spawn them.

Player → Animation

The player module controls what animation plays for the model and when.

Enemy → Animation

The enemy module controls what animation plays for each enemy and when (alongside state changes).

Player → Collision

The player is collidable and responds to collision events.

Enemy → Collision

The enemy is collidable and responds to collision events.

Enemy → AI

The enemy is controlled by the AI task and interacts with other AI elements.

Turret → AI

The turret is controlled by the AI task and interacts with other AI elements.

Turret → Collision

Turrets are collidable and are added to the collision list on creation.

Orb → Collision

Orbs are collidable and respond to collision events.

Projectile → Collision

Projectiles are collidable and respond to collision events.

Bidirectional Utilization**Animation ↔ Renderer**

The animation system relies on the shader implementation within the rendering engine and conversely the renderer relies on the animation system in order to load and display animated meshes.

Scripting Interfaces**Camera ↔ Script Engine**

The camera can be controlled through scripts.

GlobalTimer ↔ Script Engine

Global time functions and timer objects are available to the script engine.

Input ↔ Script Engine

Input checking is exposed to the script engine.

Sound ↔ Script Engine

Sound playback is exposed to the script engine.

Level Manager ↔ Script Engine

Level management functions are exposed to the script engine.

HUD ↔ Script Engine

The HUD can be updated through the script engine.

Menu Manager ↔ Script Engine

As state above, half of the camera module deals with high-level transforms available to

Kernel Tasks**Kernel → Rendering Engine**

The scene manager is part of the kernel's task list.

Kernel → Animation System

The animation system contains a kernel task to handle all animation tweening and updating.

Kernel → Texture Manager

The texture manager is a kernel task that makes sure all textures are unloaded on shutdown.

Kernel → Particle System

The particle system module contains a kernel task that handles all particle updating.

Kernel → HUD

The heads-up display contains a kernel task that it uses to draw itself to the screen.

Kernel → Scripting Engine

The scripting engine is a kernel task that executes scripts during its update phase.

Kernel → Sound Manager

The sound manager is a kernel task that makes sure all sounds are unloaded on shutdown.

Kernel → Input

The input module is a kernel task that refreshes the input devices' status during its update phase.

Kernel → Menu Manager

The menu manager is a kernel task that is responsible for rendering the menus (if applicable) and processing of any menu-specific scripts during its update phase.

Kernel → Logging Facility

The logging facility dumps all buffered log writes every update cycle to minimize disk activity.

Kernel → Memory Manager

The memory manager has a task that performs garbage collection when executed.

Kernel → Collision

The collision module has a kernel task which incrementally updates the collision status of all objects to allow for performance throttling.

Appendix A

Memory Map

128M Available System Memory

50M System Memory

2M Executable
32M Loaded scripts/Lua overhead
16M Game Objects

64M Available Video Memory

53M Video Memory

5M Shader programs
32M Textures
16M Vertex Data

Integration Plan

Integration Officer: Josh Stefanski

Intervals:

Non-milestone: Weekly, every Thursday or Friday (schedule-dependant).

Milestone: No less than three days before milestone date.

Module Completion Requirements:

A module is ready for integration when its interface has been finalized according to the proper specifications, its internals have been completed, and the developer has already tested integration with their own copy of the game source and no immediate bugs or crashing has occurred.

Integration Steps:

1. Make sure the module meets the Module Completion Requirements listed above.
2. Code is reviewed by at least two other team members for basic bug and logic checking.
3. After module is reviewed, it is submitted to the integration officer or the acting integration officer to be merged with the code base.
4. If any conflicts are encountered, see the procedure entitled 'Conflicts' under 'Source Control Guidelines'.

Decision Making Process:

If integration causes previous functionality to start crashing, the team has one full workday to try and fix the problem. In the instance that this integration crash comes within two days before a milestone, we will immediately rollback to the previous build and save a copy for the milestone before trying to fix the new crashes.

After one full day of work time, the team must decide whether or not it is time to rollback to a previous build. First there will be a vote to determine if there is any conflict on whether or not it is necessary. If there is conflict, the final decision is on the Integration Officer (Josh Stefanski), but it is also his judgment call to determine if any amount of additional work time (be it a few hours or a full day) may be allocated to the problem.

The maximum amount of debugging time that can pass before reevaluating the situation is one day of work time, or 8 hours.

Backups:

Code backups are performed in three categories: Incremental, Full, and Milestone.

Incremental: These backups are performed just before the nightly builds and only archive files and associated changes since the previous incremental backup (usually the previous night).

Full: Full backups are performed after integration on a weekly basis. All code is archived regardless of whether or not it was modified since the last full backup.

Milestone: Milestone backups are performed after milestone integration and include full source, assets, and binaries for the milestone.

Source Control Guidelines:

Do NOT check-in untested or incomplete code: Partial modules, changes to existing modules, or other miscellaneous bits of code are **never** to be integrated.

Checking-out code (short-term): Short-term checkouts are no longer than an hour and ideally around a half hour in duration. Please keep file counts to a minimum of only what is being worked on, not what could be worked on.

Checking-out code (long-term): Long-term checkouts are longer than an hour and are typically done during integration. When checking out a large amount of code, do not check out the code base directly. Instead, make a local working copy of the entire copy of the code base and modify that instead. Then when ready to integrate back into the repository (assuming all completion requirements have been met), check out the source tree, overwrite with your local working copy, and then submit changes. This will prevent needless check-out/check-in dependencies.

Conflicts: If any conflicting files are incurred during integration, the files in question are returned to the author(s) and the conflicting lines are corrected, and then re-integrated. If any interface changes occur during this period, they must be checked that they do not break the code in additional places.

Version Commenting: When submitting anything to the source control, please include at least a brief description of what was changed, even if it was a 'typo' or quick bug fix. This helps when roll-backs need to be made later.

Testing Plans

Testing

- All group members will take part in bug testing.
- The game will be divided into sections (such as floor 1 & 2, floor 3 & 4, boss fight, etc) and each member will be in charge of testing specific sections.
- When a bug is found by a group member it will be assigned to whoever is most appropriate for the task (see the following section).

Bug Reports

- Whoever finds a bug must fill out a report and give it to the Technical Lead (Josh Stefanski), who will determine if the bug is repeatable and must be fixed.
- After the Technical Lead determines its priority, the Project Lead (Jamie Cannon) will schedule the bug fix for a specific person at a set date.
- Reports will follow the following format:

Identification: [Number]

Summary: [One to three sentences of description that summarizes the bug itself.]

Description: [Complete and detailed account of why this bug is important or needs fixing. For example if it results in a character being stuck in a wall, explain whatever visual effect accompanied the stick. Did it disappear or flicker? Etc.]

Steps to Reproduce: [Numbered list of how exactly to recreate this bug. Include any relevant details. Example:

1. Location: Level 2 of Mars, room 3.
2. Character state when it occurred: Any relevant information such as health, equipped gun, etc.
3. From here, list each step to reproduce the same effect.]

Severity: [This field will be filled in by the Technical Lead. If the bug is not repeatable, it should be marked as such here – this is the lowest priority, obviously. The highest priority is Game Crash. In between, the numbers 1-5 will be suitable where 1 is the highest, right after Game Crash.]

Build Version: [The current version of the game where the bug was found.]

Bug Assignments

- Whenever possible, the module author will be assigned to repair the bug.
- If more than three bugs are determined to be inside one module at one time, therefore too many for one person to fix, the Technical Lead will be assigned to assist.
- If the bug is found to be affecting or possibly caused by more than one module, and the modules were written by different people, one of the module authors and the Technical Lead will be assigned to the bug. If the bug is inside more than one module, and they were all written by the same person, the Technical Lead may be assigned anyway to ease the process.
 - This will be a case-by-case situation. If the bug is not fixed within two days, additional help will be assigned to it. The module author may request help at any time from any team member.
- Following reception of the bug report from the Technical Lead, the Project Lead will assemble the bug reports into an Excel sheet. They will be organized by priority and completion status, and also contain information on who is assigned to the bug and the date they began.
 - Priorities follow the same format as those found on the Bug Report itself – Game Crash (highest), 1-5, Non Repeatable (lowest).

Appendix B

Game Folder Hierarchy

