Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

Spring 5-20-2016

Scare Tactics

Tiago Martines
txm7803@rit.edu

Gabriel Ortega
go4113@rit.edu

Karan Sahu
ks6332@rit.edu

Lucas Pereira Vasconcelos
lpv1569@rit.edu

Henrique Silva Chaltein de Almeida
hxs1151@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Part of the Computer and Systems Architecture Commons, and the Game Design Commons

Recommended Citation

Martines, Tiago; Ortega, Gabriel; Sahu, Karan; Pereira Vasconcelos, Lucas; and Silva Chaltein de Almeida, Henrique, "Scare Tactics" (2016). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Rochester Institute of Technology

B. Thomas Golisano College of
Computing and Information Sciences

Master of Science in Game Design and Development

Capstone Final Design & Development Approval Form

Student Name: **Gabriel Ortega**

Student Name: **Henrique Silva Chaltein de Almeida**

Student Name: **Karan Sahu**

Student Name: **Lucas Pereira Vasconcelos**

Student Name: **Tiago Martines**

Project Title: **Scare Tactics**

Keywords: **Asymmetrical Gameplay, Haunted House, C++**

Jessica Bayliss, Ph.D.
Committee Co-Chair

Elouise Oyzon
Committee Co-Chair

Chris Cascioli
Committee Co-Chair

Ian Schreiber
Advisor

Owen Gottlieb, Ph.D.
Advisor

Cody Van De Mark
Advisor

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

Scare Tactics

By

Gabriel Ortega

Henrique Silva Chaltein de Almeida

Karan Sahu

Lucas Pereira Vasconcelos

Tiago Martines

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

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

May 20th, 2016

Acknowledgements

We would like to thank our committee members, the School of Interactive Games and Media, our friends and every person who has helped us with designing, developing and testing this project.

This work was supported in part by a grant from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Brazil.

Executive Summary

It is the purpose of this document to describe the design and development processes of *Scare Tactics*. The game will be discussed in further detail as it relates to several areas, such as market analysis, development process, game design, technical design, and each team members' individual area of background research. The research areas include asymmetrical game design, level design, game engine architecture, real-time graphics, user interface design, networking and artificial intelligence.

As part of the team's market analysis, other games featuring asymmetric gameplay are discussed. The games described in this section serve as inspirations for asymmetric game design. Some of these games implement mechanics that the team seeks to emulate and expand upon in *Scare Tactics*.

As part of the team's development process, several concepts were prototyped over the course of two months. During that process the team adopted an *Agile* methodology in order to assist with scheduling, communication and resource management. Eventually, the team chose to expand upon the prototype that became the basis of *Scare Tactics*.

Game design and technical design occur concurrently in the development of *Scare Tactics*. Designers conduct discussions where themes, settings, and mechanics are conceived and documented. Mechanics are prototyped in *Unity* and eventually ported to a proprietary engine developed by our team. Throughout the course of development, each team member has had to own an area of design or development. This has led to individual research performed in several areas, which will be discussed further in this document.

Table of Contents

1 Introduction 1

2 Game Genre Background and Market Analysis 3

3 Game Development Process 8

- 3.1 Prototype Phase 8
- 3.2 Team Management 8
- 3.3 Task Management 9
- 3.4 Schedule 10
- 3.5 Pipeline 10
- 3.6 Design Process 11

4 Game Design 14

- 4.1 Summary 14
- 4.2 Introduction 14
- 4.3 Asymmetric Gameplay 15
  - 4.3.1 Explorer Gameplay 16
  - 4.3.2 Ghost Gameplay 24
  - 4.3.3 The Light Mechanic 28
  - 4.3.4 Gameplay Influences from Other Games 29
- 4.4 Level Design 40

5 Technical Design 43

- 5.1 Tools 43
  - 5.1.1 Visual Studio 2015 43
  - 5.1.2 Unity 43
  - 5.1.3 Proprietary Tools (Debugging) 43
- 5.2 Game Engine Architecture 48
  - 5.2.1 Components / Policies 49
  - 5.2.2 Template Specialization / Inheritance 49

5.2.3 Vertex Example 50
5.3 Class Breakdown 50
5.3.1 BaseSceneObject 50
5.3.2 BaseComponent 51
5.3.3 Explorer Example 52
5.3.4 Object Factory 54
5.4 Graphics 55
5.4.1 Shadow Mapping 55
5.4.2 Hardware Instancing 55
5.4.3 Lighting Deferred Rendering (Point lights / Spot lights) 56
5.5 Skeletal Animation 57
5.5.1 State Based Animations 58
5.6 User Interface 58
5.6.1 Font rendering 60
5.6.2 Debug UI 61
5.7 Collision Detection 61
5.7.1 Template Colliders 62
5.7.2 Collider Components 62
5.7.3 Bounding Volume Hierarchy 62
5.7.4 Culling 63
5.8 Networking 64
5.9 Artificial Intelligence 65
5.9.1 Motion Planning 66
5.9.2 Decision Making 67
5.10 Optimizations 70
5.10.1 Memory Management / Static vs Dynamic 71
5.10.2 Packet Construction 72
6 Asset Overview 74
7 Play Testing and Results 79
7.1 Internal 79
7.2 Public 79

7.3 Game Developer's Conference (GDC) 80
7.4 RPI GameFest 2016 / ImagineRIT 2016 81
7.5 Result 83
8 Post Mortem 84
8.1 Successes 84
8.2 Improvements 85
8.3 Future Work 85

List of Figures

Figure 1.1 - Scare Tactics splash screen. 2

Figure 2.1 - Four Hunters against One Monster. 3

Figure 2.2 - In-Game View of the General. 4

Figure 2.3 - Player View in Monaco. 5

Figure 2.4 - Dungeon Master vs Three Heroes. 6

Figure 2.5 - Warriors Fighting a Monster Possessed by the Shadowlord. 7

Figure 4.1 - The 3 Explorers. From left to right: Trap Master, Professor and Sprinter. 16

Figure 4.2 - Generator being captured (left) and light cannon (right). 17

Figure 4.3 - The Professor Class. 18

Figure 4.4 - The Sprinter Class. 20

Figure 4.5 - The Trap Master Class. 21

Figure 4.6 - Lights affecting the Ghost. The Ghost cannot spawn minions on orange/yellow areas and any minion inside those areas is slowed. 28

Figure 4.7 - The Legend of Zelda: A link between worlds, title screen. 31

Figure 4.8 - The Legend of Zelda: A link between worlds, camera. 31

Figure 4.9 - Monaco: What's yours is mine, title screen. 32

Figure 4.10 - A level from Monaco. 33

Figure 4.11 - Dungeon of the Endless, title screen. 34

Figure 4.12 - Players in different sections of the map (left) and enemies spawning from several locations (right). 35

Figure 4.13 - Bastion, title screen. 35

Figure 4.14 - The Kid aiming his spear at the mouse cursor. 36

Figure 4.15 - Dungeonland, title screen. 37

Figure 4.16 - Dungeonland's Dungeon Master mode. 38

Figure 4.17 - Prototype Level. 40

Figure 4.18 - First Level for Ghost vs Man theme. 41

Figure 4.19 - Scare Tactics Recent Level. 42

Figure 5.1 - Usage example of the CONSOLE_COMMAND macro. 44

Figure 5.2 - Console Window filtered by the keyword Warning. 44

Figure 5.3 - Console Window context menu displaying Clear and Copy to Clipboard commands. 44

Figure 5.4 - Trace messages being displayed in the Console Window. 45

Figure 5.5 - Trace messages being displayed on Visual Studio Output Window. 45

Figure 5.6 - Variable values being debugged without Trace Window. Usage (left) and output (right). 46

Figure 5.7- Variable values being debugged with Trace Window. Usage (left) and output (right). 46

Figure 5.8 - Line drawing macro definitions. 47

Figure 5.9 - Grid rendering method using TRACE_SMALL_BOX to draw grid data on the game screen. Usage taken from Scare Tactics codebase. 47

Figure 5.10 - Grid data being displayed with use of a line drawing macro. Example taken from Scare Tactics development build. 48

Figure 5.11 - BaseSceneObject Class Diagram. 51

Figure 5.12 - BaseComponent Class Diagram. 52

Figure 5.13 - Explorer Class Diagram. 53

Figure 5.14 - Usage of Factory iterator. 54

Figure 5.15 - Object registration. 54

Figure 5.16 - Object creation. 55

Figure 5.17 - Object destruction. 55

Figure 5.18 - G-Buffer is composed of several textures, including a normal texture (top left). Diffuse texture (top right). A lighting texture (bottom right) is created using the normals and positions. The final image (bottom left) is composed using the diffuse and lighting textures. 57

Figure 5.19 - Different UI elements on different alignments. 59

Figure 5.20 - Radial fill being used to indicate the cooldown of an ability. The math is part of the shader that render sprites. 60

Figure 5.21 - Linear fill being used for Ghost's mana bar. 60

Figure 5.22 - UI rendering based on world space coordinates. 60

Figure 5.23 - Class diagram for collision detection system. 61

Figure 5.24 - Bounding Volume Hierarchy. Root Quadrants are traced in red. All other colliders are a child of one or more quadrants. 63

Figure 5.25 - Typical structure of a decorator task. 69

Figure 5.26 - Subtree declaration pulled from Scare Tactics codebase. 70

Figure 5.27 - Imp Behavior Tree assembled from various subtrees. Example pulled from Scare Tactics codebase. 70

Figure 5.28 - Intel CPU architecture. 72

Figure 5.29 - Different packet types. 72

Figure 5.30 - Packet structure. 73

Figure 6.1 - Character Mood Board. 75

Figure 6.2 - Environment Mood Board. 75

Figure 6.3 - Professor Concept Art. 76

Figure 6.4 - Sprinter Concept Art. 76

Figure 6.5 - Trap Master Concept Art. 76

Figure 6.6 - Environment Concept Art. Entrance hall (top left). Master bedroom (top right). Library (middle). Generators and light-cannon (bottom left). Bathroom (bottom right). 77

Figure 6.7 - UI icons. 78

Figure 7.1 - RPI Playtest. 82

Figure 7.2 - RPI Award. From left to right: Henrique Chaltein, Gabriel Ortega, Lucas Vasconcelos, Karan Sahu, Tiago Martines. 82

List of Tables

Table 3.1 - Most prominent group roles.......... 9

Table 3.2 - Major milestones. 10

Table 4.1 - Minion types and stats. 26

1 Introduction

Asymmetry in games exists in many forms. Turn-based games exhibit asymmetry in that one player goes first and the other second. One form of asymmetric gameplay involves assigning unique mechanics to each player. An example of this can be seen in games such as *Gauntlet* (1985), where players each select separate avatars each of which possess a distinct skill set. Another form of asymmetric gameplay is the concept of *one versus many*. In this type of game one player competes against many other players.

Asymmetric games have unique properties when compared to symmetric games. Due to offering separate experiences, an asymmetric game can appeal to a wider audience than a purely symmetric game. However, these types of games are more challenging to balance via purely numerical methods and must rely on play test data.

Although not a particularly new concept, asymmetric gameplay is an interesting pseudo-genre and serves as the core concept behind *Scare Tactics* (Figure 1.1).

Figure 1.1 - Scare Tactics splash screen.

Scare Tactics is a multiplayer game featuring asymmetric play, developed by *So Close Games*. It takes place in a haunted house where a group of *Explorers* face off against a *Ghost*, whom is in control of the house. A group of up to three people can play as *Explorers*, while one person can play as the *Ghost*. The *Ghost* and *Explorer* each have unique gameplay mechanics. The gameplay for the *Ghost* character is based on the Real Time Strategy genre. He or she can spawn and manage units, as well as manipulate the house on a global scale. The *Explorer* gameplay is based on top down action adventure. The *Explorer* can perform melee attacks, acquire power-ups and manipulate the environment on a local scale. Both characters use their unique skills and mechanics to thwart each other. The *Explorers* must exorcise the haunted house in order to defeat the *Ghost*, and the *Ghost* must disable all *Explorers* by diminishing their health. It was the goal of our team to design and develop *Scare Tactics* as an asymmetric gameplay experience.

2 Game Genre Background and Market Analysis

Our game is focused on providing players with an asymmetric '1 vs many' gameplay experience. We are looking at games that provide a similar experience. Some of the games that we have come across are *Evolve, Heroes and Generals, Dungeonland,* and *Shadow Realms.*

In Evolve, one player plays as the monster while the other four play as hunters (Figure 2.1). Each of the two sides tries to kill the other. It starts out with the monster underpowered and the hunters chase it to take it down. Over time the monster becomes stronger and a force with which to be reckoned. For the majority of the game, the monster is running away from the hunters and trying to catch prey to level up and become stronger. The monster does not have any support while the hunters can be revived by other hunters. From our perspective, the monster does not feel all that overwhelming. This is the opposite of what we are trying to achieve. We want the *Explorers* to be scared of the *Ghost* from the outset. If they are careless and within reach of the monsters, they should be taken down.

Figure 2.1 - Four Hunters against One Monster.

Heroes and Generals is a massive multiplayer battle set during World War II where heroes have a first person point of view and are directly involved in the war. The generals have a map to decide where the battles take place. They could strategize to attack a city to capture it or defend against attacks from the opponent (Figure 2.2). The General is never directly involved with a battle taking place in a city. He does not have detailed information about the troop movement and their actions.

Figure 2.2 - In-Game View of the General.

We do not want this with our *Ghost* player. His view is similar to the general's but is limited in terms of scope and has much more information on what the players on the ground are doing. We want the *Ghost* player to see the *Explorer's* movements and actions in the level (Figure 2.3) instead of a general description such as running or populating, in case of *Generals and Heroes*. We want him to have detailed visual information on every *Explorer*, whenever the gameplay allows, which would allow him to strategize accordingly and be directly involved with the "battle."

Figure 2.3 - Player View in Monaco.

Dungeonland is another Player vs Player asymmetric game very similar to ours in terms of interaction between the two player types (Figure 2.4). Three heroes battle against swarms of monsters that are spawned by a dungeon master. The goal of the heroes is to get to the end of the dungeon and defeat the boss that is controlled by the dungeon master. If they die, the dungeon master wins. The ghost in our game has a very similar role to the dungeon master. He can spawn his minions, place traps and capture the players. We are aiming for a mechanic where the *Ghost* can possess an *Explorer* and cause him to have hallucinations. The *Explorer* would see things that are not actually present. This could lead the *Explorer* into a trap or mistake another *Explorer* for a minion of the *Ghost*. This could harm the *Explorers* as the game will have friendly fire. We also want the *Explorers* as well as the *Ghost* to be able to interact with the environment, to be able to use it to their advantage or to hinder their enemy.

Figure 2.4 - Dungeon Master vs Three Heroes.

Shadow Realms is a fast paced third - person action game where a team of warriors fight against an evil called the shadowlord (Figure 2.5). Their goal is to destroy the other player, unlike *Dungeonland* where the heroes can only win by getting to the end of the level. The shadowlord can spawn monsters that attack the warriors and create traps that get activated when a warrior is close to it. These two mechanics relate back directly to our game. Similar to *Dungeonland*, the shadowlord can possess other monsters and directly control them. We have a hallucination mechanic that is triggered by possessing an *Explorer*. The shadowlord can also create a copy of himself and disguise it as the warriors. He cannot be identified until he receives damage. We also wish to focus on the idea of hidden information. We want to give extra information to the *Ghost* to make him or her feel dominant. The additional knowledge would give an advantage over the *Explorers*, allowing a better strategy for placing traps and minions.

Figure 2.5 - Warriors Fighting a Monster Possessed by the Shadowlord.

The games currently in this genre are mostly action-oriented and include some elements of strategy. The player fighting against a team of players is outnumbered, but does not necessarily feel extremely evil and overpowering. Scare Tactics intends to take the action setting and add Real Time Strategy (RTS) and stealth elements. We want to reinforce the “1 vs many” gameplay experience. We want the Ghost to feel powerful, to feel like the ruler. On the other hand, the Explorers cannot see the Ghost as s/he has no physical presence on the map. We want the Explorers to tread carefully and be afraid of the Ghost. The player assuming the role of the Ghost has an overarching view of the map and is in charge of the environment, thus enforcing a global force. The players in control of the Explorers have a hyper-localized view of the map and are not as aware of the Ghost’s actions nor the surprises that lie ahead of them. In addition, we want our co-op gameplay to be distributed. We want to avoid the Explorers clustering all the time. We want them to split up in order to cover more ground and make it more difficult for the Ghost to catch them.

3 Game Development Process

3.1 Prototype Phase

Our development process started in September 2015, with a 2-month ideation period. During this time, the group focused all of its efforts into creating a unique prototype per week, covering different game styles, themes, mechanics and feelings. Prototypes were put to test with other individuals to help with the evaluation process. This extended ideation phase also played an important role in defining our development process – after every weekly cycle we met for a post-mortem assessment, iterating and polishing the process so it became suitable for our team. More importantly, by the end of this ideation period and the following selection phase, we had a solid game idea that was promptly embraced by all members.

3.2 Team Management

With the selected game idea, we started focused development by November 2015. Initially, we decided to use the some of the techniques often employed for software development via the *Scrum* methodology. In addition, each member of the team had been assigned a couple of roles according to individual areas of expertise and interest. Although all individuals are working on every aspect of the game, picking some roles and leaders helps to address conflicts and create guidelines more efficiently. The most prominent roles are listed on Table 3.1.

Table 3.1 - Most prominent group roles.

Scrum Roles		Collaborators	
Product Owner	Karan	Concept Artist	Jeannette Forbes
Scrum Master	Lucas/Gabriel	Concept Artist	Katherine Harrison
Backlog Maintenance	Henrique	Concept Artist	Felipe Yoon
Field Expertise Leaders		Rigger/Animator	Dillon Guscott
Design	Henrique / Karan	3D Modeler	Kerong Fu
Engine Architecture	Gabriel	3D Modeler	Steven Cerqueira
Graphics Programming	Gabriel	3D Modeler	Robert Marsh
AI Programming	Lucas	3D Modeler	Jesse Florio
UI Programming	Tiago	3D Modeler	Ziyun Peng
Network Programming	Tiago	3D Modeler	John David Satriale
Art	Karan	Audio Designer	Kedar Shashidhar
Organization Roles		Audio Designer	Rick Scott
Process	Tiago		
External Resources	Karan		

3.3 Task Management

We started by selecting a tool to aid us with controlling tasks, bugs, documentation and other artifacts that are employed during our process. We selected *Redmine*, an open-source web-based project management and issue tracking tool, mainly because this tool is free to use and allows total customization. For our group, this customization level was especially useful to trim down the features that are usually included in a fully-fledged Scrum environment, but that we considered detrimental to our process given our team size and specific goals.

Moreover, *Redmine* also integrated seamlessly to the version control system we are using, *Git*, allowing commit and tags to be directly linked to certain tickets, either bugs or feature requests. *Redmine* also allows our stockholders, mostly our committee members and other interested faculty, to login and follow up with our tickets, sprint burndowns, released versions and documentation.

3.4 Schedule

With the process tools and methods set, we started to sketch our production schedule. We broke down the timeline into six major milestones, as shown on Table 3.2.

Table 3.2 - Major milestones.

Milestone	Details
Ideation	Deadline: November 1st, 2015
	Main activity: Weekly prototypes
	Output: A solid game idea
Proposal	Deadline: December 16th, 2015
	Main activity: Design; Technical Research
	Output: Game proposal; Playable Unity prototype
Winter break	Deadline: January 24th, 2016
	Main activity: Feature set definition; Core implementation; Research
	Output: Major mechanics implemented in Unity; Framework for C++
GDC Build	Deadline: March 10th, 2016
	Main activity: Porting; Code optimizations; Balancing; Assets implementation
	Output: Public playable C++ game
Imagine RIT / RPI	Deadline: April 20th, 2016
	Main activity: Aesthetical improvements; Balancing; Bug fixing; Documentation
	Output: One fully playable game mode; Documentation
Final	Deadline: May 18th, 2016
	Main activity: Polishment; Bug fixing;
	Output: Final game; Pitch and Presentation;

3.5 Pipeline

Following Scrum practices, every week we arranged a sprint planning meeting to specify which tasks were immediately more relevant towards the milestone outputs. Each sprint produced a minor version of the game, e.g., 0.2 or 0.3. A major milestone release was identified by a major version number, e.g., 1.0 or 2.0. With this methodology, we wanted to ensure there would always be a current build readily available for playtest or exposition.

Moreover, our sprint planning tries to schedule most individual tasks in an efficient manner. Design tasks are always followed by a prototyping task, allowing a playtest task to come right after it.

After playtesting, the feature is scheduled for implementation. By pipelining those tasks accordingly, we want to ensure there are fewer roadblocks between design and development phases.

Most of the design decisions involved using *Unity* as prototyping tool. Thereby, we devoted much of our planning to making tools and policies to export content, levels, configuration and other assets from *Unity* into our proprietary engine. With this solution, we allowed the design team to work faster and as independently as possible from the development team. We believe this decision greatly improved the experience we had while developing *Scare Tactics*, as it allowed every member of the team to focus on and improve their areas of interest. More about our prototyping and development environments is detailed on section 5.

3.6 Design Process

By the end of the prototyping phase of our development, we created 6 unique prototypes. Out of the 6, we voted to develop the prototype titled *Shutter* into our capstone project. The theme of the prototype was prison and considering our asymmetrical gameplay, we felt it might be difficult to expand this theme into a long term project with interesting mechanics. So we went back to the drawing board to come up with themes that would fit better into our asymmetrical gameplay style. The most promising themes from a long list were *Nature vs Industrialism* (a worker expanding his city versus the Spirit of Nature reclaiming his territory), *Magic vs Technology* (A scientist and a Wizard fighting over whether theirs is the best method to help their village) and *Ghost vs Man* (Ghost Squad trying to exorcise a ghost house.) As a group we settled on *Ghost vs Man* as it appealed to the majority of us and we felt it had the potential to grow.

The next big question we needed to answer was whether to include a narrative and create a single player campaign or focus on multiplayer. We did not have a dedicated story writer and none of us had done it earlier. A single player campaign also meant we would have to create a long enough gameplay and have an ending, in the game and not just on paper. This seemed challenging as we opted to create

the game in C++. We had very limited time and were a small group of 5 people. Thus, we changed our direction and decided to create a multiplayer only version with a few game modes. Each mode would have a short narrative within itself, trying to give the game a bit more depth.

We opted for the approach to overscope the design and cut down the features that weren't as exciting and created a priority list for the ones we wanted to implement in the game. Due to this approach we created 4 different game modes, but only managed to implement one, Landmark.

1. Landmark - activate the weapon in the middle of the haunted environment to kill the *Ghost*
2. Hostage - find and guide the hostages trapped in a haunted amusement park to the exit
3. Escape - escape a haunted junkyard before the *Ghost* kills every *Explorer*
4. Escort - guide a scared priest to the heart of the haunted house to exorcise the *Ghost*

Four *Explorer* classes:

1. Scout - evolved into *Sprinter*
2. Support - evolved into *Professor*
3. Offensive Long Range - evolved into *Trap Master*
4. Offensive Short Range

Thirteen minions:

1. Basic Melee - evolved into *Imp*
2. AOE Bomber - evolved into *Abomination*
3. Ambusher (Defender) - evolved into Flytrap
4. Basic Projectile
5. Triangle Squad
6. Attach

7. Poison
8. Barrier
9. "Transporter (Worm)"
10. Bull
11. "Trespasser (Ceiling Goop)"
12. Ambusher (Chaser)
13. Hulk

4 Game Design

This section describes Scare Tactics game design. It will demonstrate all design decisions made for the game as well as the reasoning behind those decisions.

4.1 Summary

Scare Tactics is a hybrid action adventure/tactical online multiplayer game focused on player versus player interaction for children aged 6 and up. The game is set in a house that is haunted by a *Ghost*. A group of three *Explorers* decide to go to said house in order to exorcise the *Ghost* that lives there. The game is a one versus many multiplayer game that provides two different experiences depending on the role the player chooses to play. The Ghost player has a slower-paced, individual and more tactical experience while the Explorers have a faster-paced, team based and action-oriented experience.

4.2 Introduction

As mentioned before, *Scare Tactics* is a hybrid action adventure/tactical online multiplayer game focused on player versus player interaction. The purpose of this game is to provide two completely different experiences to the players depending on which role they decide to play. These roles are separate and are meant to not directly influence one another. These roles are the *Ghost* and the *Explorers*.

The game takes place on a creepy old mansion where the Ghost lives. The Ghost has absolute power inside the mansion, being able to summon monsters (called *Minions*) at will, create illusions and telepathically manipulate objects. The Ghost enjoys ruling over the mansion and will attack anyone who dares to disturb its territory.

The Explorers are a group of thrill-seekers that travel around the world chasing ghosts and exorcising them. They are the *Professor*, the *Sprinter* and the *Trap Master*. They use the light-based technology developed by the Professor in order to fight ghosts and their minions around the world.

Each explorer has their own unique skills fulfill a specific role on the team. The game starts when these Explorers find the Ghost's haunted mansion and decide to go inside and exorcise the Ghost.

In *Scare Tactics* two distinct roles are put against each other. The Ghost, although all-powerful, is alone and has to manage its minions and the whole mansion while the Explorers, significantly weaker by themselves, must use each character's unique skills and teamwork.

Despite the creepy atmosphere and Haunted Mansion theme, *Scare Tactics* sports a more cartoony style and is geared towards children from age 6 and up. Playtests showed a surprisingly good response from children around that age, as discussed on section 7.

The next few sections will go into deeper detail on the design decisions made during the development of *Scare Tactics*.

4.3 Asymmetric Gameplay

Asymmetric games are those in which different player roles play differently. This difference in play can come from different mechanics, tasks, visuals, advantages, disadvantages, etc. attached to these roles. The asymmetric design of *Scare Tactics* is focused on having different mechanics and tasks attached to the different roles. The gameplay experience attached to each role is meant to be played notably different from the other. The team's objective was to bring together these two different game experiences and merge them into one.

Merging two game experiences is not a simple task. Both game experiences have to be compelling and unique on their own without being independent from the other. The gameplay loops of both experiences have to be distinct, but still influence one another. The core of this challenge was striving for this balance between independence and interconnectivity.

When playing as an Explorer in *Scare Tactics* the player will have a game experience focused on moment to moment decision making with an emphasis on action and teamwork. When playing as the Ghost players will have an experience focused on long-term decision making with an emphasis on

strategy and resource management. The next sections will discuss each of these experiences as well as the differences between them.

4.3.1 Explorer Gameplay

As mentioned before there are three Explorer classes: the *Professor*, the *Sprinter* and the *Trap Master* (Figure 4.1). Each one of these classes fulfills a specific role on the Explorer's team and has unique skills that help players fulfilling that role. This section will first discuss the general gameplay shared by all explorers and then discuss each one of the explorer classes separately, focusing on their different skills and mechanics.

Figure 4.1 - The 3 Explorers. From left to right: Trap Master, Professor and Sprinter.

General Gameplay

The Explorer's objective in Scare Tactics is to exorcise the Ghost. In order to achieve that goal they need to power their *Light Cannon*, shown as a concept in Figure 4.1 and during gameplay on Figure 4.2. However, the light cannon takes an enormous amount of energy to be powered up. To generate enough power, the explorers must find 3 generators hidden on the house, one in each floor, and redirect their energy to the light cannon. Once powered up, the Explorer's can activate the Light Cannon, defeat the Ghost and win the game.

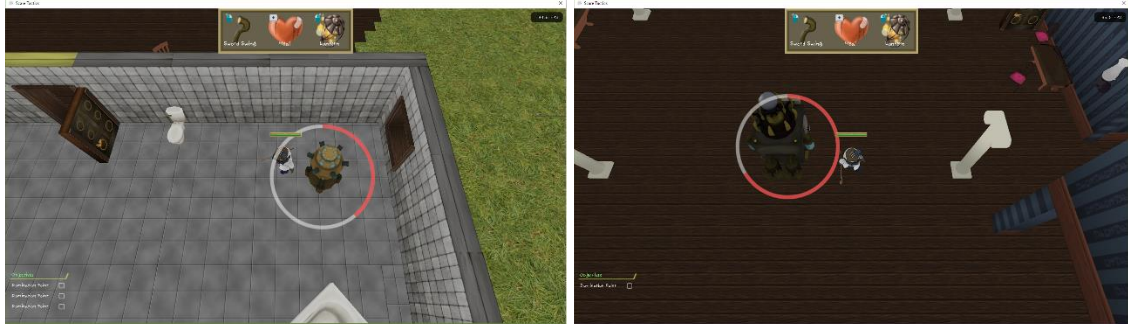

Figure 4.2 - Generator being captured (left) and light cannon (right).

Gameplaywise the *Generators* and the *Light Cannon* work as a typical capture point usually present in multiplayer "King of the hill" style games. A capture point is a finite area where players must stand inside for a certain amount of time in order to score points. When the game starts all 3 generator capture points are available and the Light Cannon capture point only becomes available once the 3 Generator points are captured. When the Explorer captures this last point, they win the game.

The Explorer role in *Scare Tactics* is the one inspired by the top down adventure genre. A lot of the general Explorers mechanics is lifted directly from conventions of this genre. All explorers have the ability to:

- **Move around the level**: One of the most basic abilities of any game. Explorers can move in all 8 cardinal directions (North, South, West, East and the 4 diagonals between them). Players cannot jump.
- **Class Attack:** The main method by which Explorers defend themselves and fight minions. Each class has its own unique attack with their own separate properties that will be discussed in the next session.
- **Drop Portable Lantern:** At any time Explorers can drop a *Portable Lantern* on the floor. Lanterns in general are light sources and, as such, they have a special effect on the Ghost and its minions. This is further discussed on section 4.3.3.

- **Turn on/off Wall Lanterns:** Explorers can turn on/off Wall lanterns if they are next to them. These Wall Lanterns are scattered through the level and are light sources similar to the Portable lanterns. As mentioned, lights and their effect on the Ghost and its minions is further discussed on section 4.3.3**.**
- **Revive a Downed Ally:** When an Explorer takes enough damage, he falls on the ground and becomes unable to fight. Another Explorer can go near his downed ally and revive him, healing a small amount of that explorers health and enabling him to fight again.

Classes and Skills

This section will detail the three Explorer Classes: The Professor, The Sprinter and the Trap Master.

The Professor Class

The *Professor* is the one that invented the technology used by the Explorers to fight The Ghost. He is small in stature and little bit portly, however he is very smart and methodical. He prefers to keep his distance and only take action if strictly necessary. Figure 4.3 shows the professor.

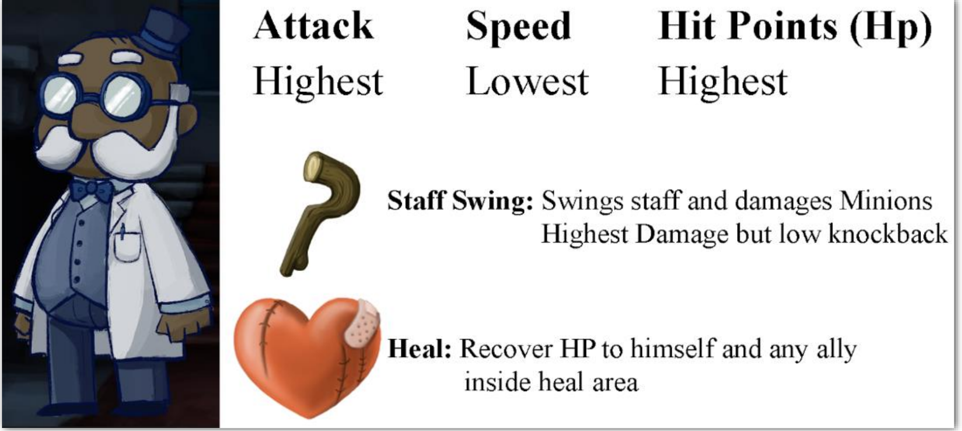

Figure 4.3 - The Professor Class.

On the Explorers team he acts as a support class. His unique skill is the ability to *heal* allies. He can create a green light healing sphere centered on himself that slowly expands outwards, healing himself and any ally that touches said sphere.

Even though he is not a fighter, he is equipped with a heavy staff that he uses to hit minions if they get too close. The staff does a lot of damage due to its weight alone, but the Professor is not strong enough to wield it with enough force to push minions.

Due to his physique, he is the slowest of all Explorers, often being left behind. To compensate he has the highest attack power and hit points. These stats, *Attack*, *Speed* and *Hit Points*; varied greatly during the development of *Scare Tactics*. He was always meant to be the slowest character but the maneuverability of the Explorers proved to be one of their core assets when it came to surviving the Ghost's attacks. Speed is such an important status that it is necessary for the slowest character to have both the highest Attack and HP.

Good Professor players will learn to stay close to the group, staying back and healing whenever necessary. More often than not, the success of an explorer team is dependent on how well the Professor player plays.

The Sprinter Class

The *Sprinter* is the assistant of the Professor. She has worked for him for a long time and is responsible for the development of some of the technology used by the Explorers. She is tall and slender, which makes her fast and agile, but not physically strong. She is not as smart as the Professor since she lacks his years of study, but she is sharp and quick witted, being able to quickly adapt to most situations she finds herself in. Figure 4.4 shows the sprinter.

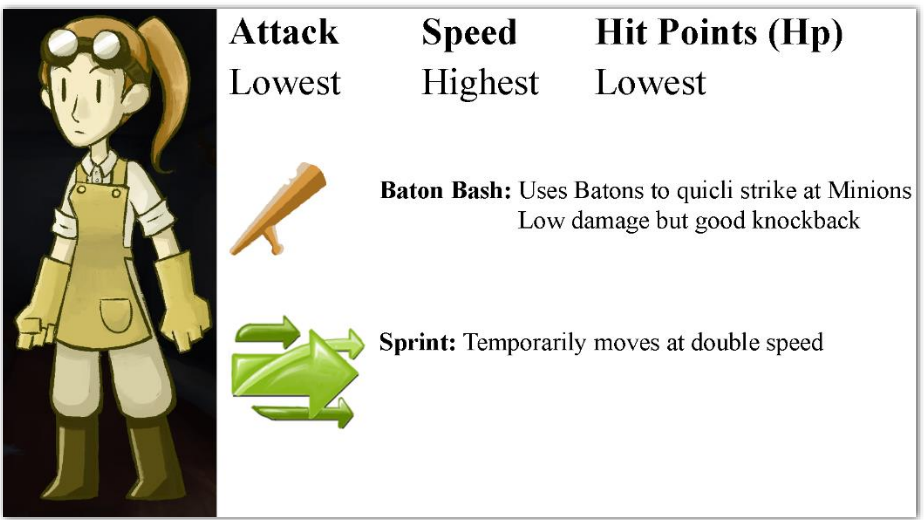

Figure 4.4 - The Sprinter Class.

On the Explorers team she acts as a Scout class. Her unique skill is the ability to S*print*, which makes her twice as fast for a limited amount of time. This is a versatile ability that can be used to either escape a bad situation or rush forward to take advantage of an opportunity.

Her weapons of choice are two Batons that she can quickly swing at Minions to cause damage. Even though each blow does low damage individually, the hits tend to pile up making her a decent damage dealer. Her attacks can push minions away and, due to her speed, Players have the choice to either pursue and finish off a Minion or retreat after each blow.

She is the fastest of all Explorers, which means that she usually rushes ahead by herself. However, her low amount of *Hit Points* makes her very fragile and a perfect target for the Ghost when she is alone. In a reverse situation than that of the Professor, her speed made her a powerhouse during the development of *Scare Tactics*. It was necessary to make her have the lowest Attack and HP; otherwise she could defeat the Ghost by herself.

Good Sprinter players will understand the value of the hit and run strategy as well as sticking close to the team. While her team heavily defends an area, she can use herself as bait to either lure powerful enemies away from her teammates while they gather themselves or to lure weak enemies into her allies for easy pickings.

The Trap Master Class

The *Trap Master* is a roguish thrill-seeker friend of the Professor that can never say “no” to an adventure. He became friends with the Professor after they stumbled on each other at the University’s library. The Professor was looking for a book about Ghosts and the houses they haunt for research purposes, the Trap Master was looking at the same book looking for an adventure. Years of chasing trouble made him very fit physically and gave him “street smarts”. He is not especially intelligent nor dumb, but he instinctively knows how to get out of a bad situation through luck and quick thinking. Figure 4.5 shows the sprinter.

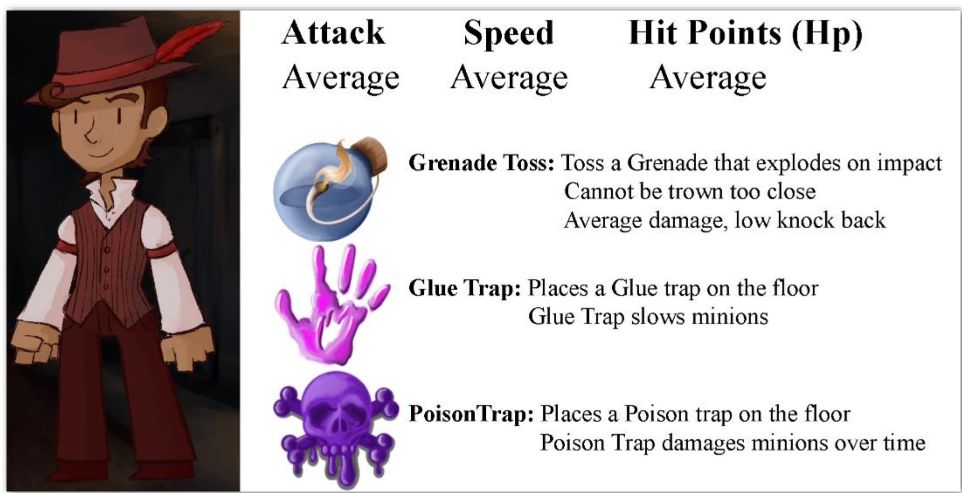

Figure 4.5 - The Trap Master Class.

His role on the Explorer’s team is to Control Space. He can place two different kinds of traps that have different effects on the minions. The *Glue Trap* can slow minions down to a higher degree than that of a light source (see section 4.3.3 for more details) and the *Poison Trap* unleashes poisonous gas that deal consistent damage to a Minion over a short amount of time. When placed, a Trap stays on the level until it is triggered. Once triggered said Trap unleashes its effect for some time and then disappears. Traps can only be triggered by Minions but their effects can harm Explorers as well.

To attack the Trap Master uses a special kind of *Grenade* developed by the Professor that only harms Minions. These Grenades explode on contact and damages all Minions hit by their explosions.

They can only be thrown at a fixed range from the Trap Master, which makes him vulnerable to minions that come too close to him.

Out of all the classes, his attack and abilities are the hardest ones to use. Both Grenades and Traps require great spatial awareness from the player and the misuse of the Trap Master skills can cause serious harm to the whole team. Therefore, he was made into the "Average" stats wise class to ease his learning curve a little bit.

Good Trap Master players will be aware at all time of where they, his teammates, his enemies and his traps are. He will use this information to his advantage. Poison traps are especially good against minions with low mobility while Glue traps can give the team a chance to counter a bad situation. They will work with Sprinters to set up ambushes for enemy minions or work with Professors to set up a powerful defensive position holding their ground using traps, heals, grenades and occasionally whacking a Minion or two with a staff.

Cooperative Gameplay

The *Explorer* aspect of *Scare Tactics* relies heavily on cooperative gameplay and requires players to communicate with each other. Cooperation and teamwork are the key to success for the Explorers, given that they can be easily overwhelmed when acting alone. The previous section detailed each Explorer class and, by looking at it, it's easy to see how they all need each other.

The *Professor* is powerful and can heal himself, but his low maneuverability makes him easy to be surrounded and overwhelmed when alone. His *Heal* ability is very powerful, but can only delay the inevitable. However, if another Explorer can help him open a way through the surrounding Minions and he uses his *Heal* ability well; they can overcome this situation together.

The *Sprinter* is very fast and her higher attack rate makes her a decent damage dealer, but she more often than not puts herself into dangerous situations. Even she is not fast enough to escape all Minions unscathed without help. All small hits she takes pile up and she will eventually succumb due

to her low HP. The Professor can help her simply by healing her wounds and enabling her to keep fighting and the Trap Master can help her by dealing the extra damage or slowing down the Minions enough that she can avoid them.

The *Trap Master* class has access to very versatile skills and can adhere to multiple strategies, but he has a fatal weak point at close range. He is very good at keeping minions at a distance but he cannot keep them all away. Eventually some minions will break through and take him down. Any other one of the other classes can cover this flaw simply by being able to attack at close range. A combination with the Sprinter tends to be more offensive while one with the Professor more defensive.

That being said it is not strictly necessary for the explorers to always keep together. Strategies that involve one player acting alone are viable, if well thought out. The player acting alone is definitely more vulnerable, but he can either act as bait to distract the Ghost or rush towards an objective; for limited time. Cooperation does not necessarily mean always being together; but rather acting together using a strategy that best fits the team's needs.

By being together, Explorers can cover up for their weaknesses and revive one another if necessary. But that also means that they will have the Ghost's full attention and forces to deal with. On the other hand, by splitting up they also split up the Ghost's resources and they can fulfill objectives faster, but they become much more vulnerable. Banding together makes the game last longer and could be interpreted as a long-term strategy, while splitting up is a high-risk high reward strategy that has potential to either succeed or fail marvelously.

All classes have the ability to survive for *some time* by themselves as exemplified in the previous paragraphs. All classes only fall after *some time*. This survival time limit could only be achieved by carefully tweaking and balancing the Explorers and Minions stats and has proven to be one of the greatest challenges when designing *Scare Tactics*.

4.3.2 Ghost Gameplay

The other player role in *Scare Tactics* is the *Ghost*. Differently from the Explorers, there is only one *Ghost*. This section will discuss the gameplay aspects of the Ghost role as well as the *Minions* and skills, called *Haunts*, available for players in this role.

General Gameplay

The *Ghost* is an all-powerful entity that rules over the Haunted House in which *Scare Tactics* takes place. No one really knows when he started haunting this mansion, or why. All that it is known is that visitors are not welcome. Should a group of adventures be foolish enough to invade this Ghost's territory they will have to face hordes and hordes of Minions.

The *Ghost's* objective in *Scare Tactics* is simple: To kill all Explorers that have invaded its house. However, the Ghost cannot interact directly with the Explorers. Instead it must summon Minions and use Haunts to either kill the Explorers, or trick them into killing each other.

As mentioned in section 4.3.1, the *Explorers* objective is to redirect three *Generators* energy to the *Light Cannon* and use it to defeat the *Ghost*. Therefore, the Ghost's objective is to defend said generators and defeat the Explorers. In order to achieve this objective, the Ghost has access to *Minions* and *Haunts*.

Minions are monsters summoned by the ghost each with their own unique status and behavior. Minions are designed so that each one has a specific use and can be considered a different tool in the Ghost's arsenal.

Haunts are the ghost skills in which he manipulates the environment or the senses of the Explorers. Save for one exception, Haunts were designed as a support system for the Minions, allowing the Ghost to have influence over their behavior and effectives in some way. The exception of the Haunts is *Imp Illusion*, which directly interferes with the Explorers and was designed to spread misinformation and doubt among them.

Both Minions and Haunts are discussed more in depth in the next section.

Gameplaywise, The Ghost's role in *Scare Tactics* is similar to that of a *Dungeon Master* in traditional pen and paper RPGs. The Ghost is the one responsible for crafting the obstacles that the Explorer's must overcome to win the game. If the Explorers succeed in clearing these obstacles, they win but if they don't, the Ghost wins.

The Ghost's gameplay is slower and more tactical when compared to the Explorer's. The ghost focus on mid- to long-term decisions as well as managing the resources available to it. The Ghost first hatches a plan, then he executes and, if something unforeseen happens, he adapts.

Good Ghost players will be aware of the consequences of their actions, as well as know which Minion or Haunt is appropriate for the situation they are currently in. Players will be able to set ambushes, trick explorers and essentially guide the game to a more desirable state while accurately managing their resources. They understand if they misuse their resources they will be helpless for a small period of time that could very well mean victory or defeat.

Minions and Haunts

As mentioned in previous sections, the *Ghost* has both *Minions* and *Haunts* at its disposal. This section will discuss the different *Minion* types as well as the *Haunts* currently available on the game.

Minions are monsters summoned by the *Ghost* to do his bidding. Each has their own unique characteristics as well as their own unique behavior. Table 4.1 contains all *Minion* types, as well as their characteristics.

Table 4.1 - Minion types and stats.

Minion Type	Imp	Abomination	Flytrap
Description	Small but vicious demons that quickly attack any Explorer they set their eyes on.	An agglomeration of flesh and viscera that attacks by exploding itself on their target.	A possessed house plant that would rather eat flesh than bask in the light of the sun.
Role	Small melee attackers. Make up the majority of the Ghost's army. By itself it poses no threat, but in numbers can be a force to be reckoned with.	Big damage dealer. It requires support in order to hit the Explorers, but the damage it does when exploding more than compensate for it.	Immobile defender. Used to guard points of interest. Can be used to create choke points or to trap Explorers in a room.
Behavior	Quickly roams around the House and immediately attacks first explorer sighted.	Slowly roams around the House. Once it sees an Explorer, start moving towards and blow itself up when in range.	Once an Explorer is close enough turns itself towards him. If the Explorer gets even closer, attacks viciously.
Attack	Average	Very High	Very High
Speed	Fast	Slower	Immobile
Hit Points	Low	High	Highest
Mana Cost	10	25	20

It is important to note that the Ghost cannot directly control *Minions.* The *Ghost* needs to keep the *Minions* behavior in mind when summoning them as well as predict what they will do. Differently from the *Haunts*, *Minions* cost mana to be summoned. Each minion has a cost associated to it, as shown on Table 4.1**.** The Ghost's mana bar slowly replenishes itself and, if the Explorer's manage to complete an objective, the Mana bar refills itself faster. This allows for more minions to be summoned as the Game progresses.

Haunts are utility skills that the ghost can use for no cost with the specific purpose of supporting the *Minions* and confusing explorers. They are:

- **Open doors:** *Minions* do not have the ability to open doors. Sometimes Explorers use this information to their advantage by locking a bunch of minions inside a room. With this *Haunt* the *Ghost* can release these captured minions when the Explorers least expect it. More experienced *Ghost* players will try to use a locked room an ambush; gathering a bunch of minions inside a locked room on purpose and releasing them when necessary.
- **Turning off Wall Lanterns:** Light sources have the ability to interfere with the *Ghost*'s summoning abilities as well as weakening minions. This is further expanded upon in section 4.3.3. Wall lanterns are one of these light sources. The *Ghost* has the ability to quite literally turn this annoyance off.
- **Imp Illusion:** This is a very different kind of *Haunt*. This haunt is not as simple as the others and it is not meant to support the *Minions*. It instead confuses and tricks the *Explorers* into attacking each other. This *Haunt* targets an explorer and, when in effect, it disguises that explorer as an Imp. However, two important things happen: The targeted *Explorer* is not aware that he is disguised as an Imp *and* the targeted explorer becomes susceptible to friendly fire. By twisting the perception of the other Explorers, the *Ghost* has the ability to trick the Explorers into killing one of their own by using this skill.

Haunts are mainly support skills available to the Ghost. However, the combination of both *Haunts* and *Minions* enable the *Ghost* to execute a variety of elaborate strategies in order to defeat the *Explorers*. Be it by using superior strategy, trickery, or simple brute force the *Ghost* is a force to be reckoned with and it is best for the *Explorers* to be aware of that.

4.3.3 The Light Mechanic

As an asymmetric game *Scare Tactics* has two distinct roles. Each role has its own gameplay completely separate from each other. But an asymmetric game must go beyond just having two different roles, but also make these roles interact with one another. One of the design cornerstones the team adopted was "the Explorers and the Ghost do not interact directly with one another, they use the environment and other indirect interactions instead". Adhering to this cornerstone heavily influenced how the Ghost's *minions* and *haunts* were designed, for example. Another main mechanic that was born from this cornerstone is the Explorers and the Ghost relation to *lights*.

In the world of *Scare Tactics* lights have the ability to weaken the Ghosts and its minions. Gameplaywise this means that the Ghost cannot spawn minions on lit areas and any minion that walks into a lit area becomes slowed, as shown in Figure 4.6.

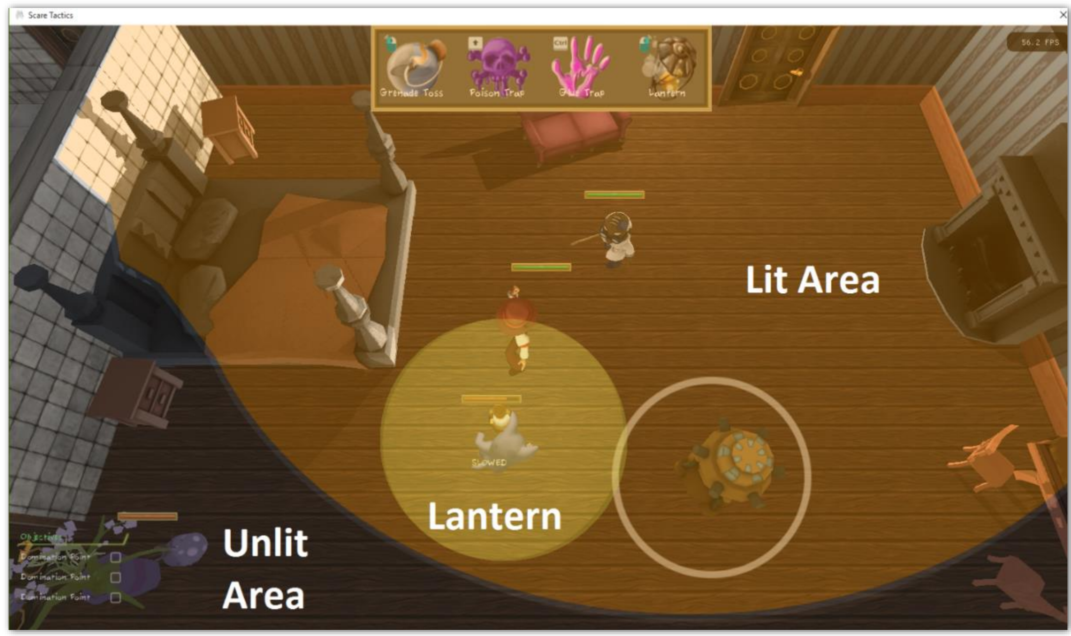

Figure 4.6 - Lights affecting the Ghost.
The Ghost cannot spawn minions on orange/yellow areas and any minion inside those areas is slowed.

Through this mechanic, the Explorers can interact with the Ghost using the environment and positioning themselves on lit areas. Lit areas become a safe haven for the Explorers since they have an

overwhelming advantage against the Ghost's minions in those areas. However, the amount of lit areas is limited. Lit areas are created by turning on *Wall Lantern*s scattered throughout the walls of the Mansion and they are placed in way so that it is impossible to light the whole mansion. The Ghost also has the ability to manually turn the wall lanterns off, creating a war of attrition between the Explorers and the Ghost.

Another way the Explorers can create lit areas is through the use of *Portable Lanterns*. As mentioned before, one skill all Explorers share is the ability to place these Lanterns on the level. Each Explorer can only place one lantern at a time and they disappear after some time, however they cannot be turned off by the Ghost. They act as a temporary defensive solution that can be used in a pinch by the Explorers.

Depending on which role the player play as, light is interpreted in different manners: for the Explorers it is a protective field that can mean the difference between success and failure; for the Ghost it is a constant annoyance that limits its power.

4.3.4 Gameplay Influences from Other Games

Scare Tactics is a hybrid action adventure/tactical online multiplayer game. As mentioned before, this means that as a hybrid game *Scare Tactics* contains gameplay experiences on both of these genres. The team looked at other games for inspiration and reference when designing the gameplay of *Scare Tactics*.

This project considers top-down action adventure games to be those in which the player controls a character with a defined set of skills exploring a world and fighting against enemies using said skills. These games use a camera placed above the player's character, facing down. The player character may or may not have their skill set defined by a character class and new skills may or may not be unlocked as the game progresses.

This project considers Tactical games to be those in which the players must summon units on a map in order to complete his objectives. It also focuses on games that run in real time, i.e. time constantly moves once the game start and there are no turns. These games usually involve resource and base management, but not necessarily. They usually have a bird's eye view camera and mouse-driven controls. All units summoned by the player may be available from the start or be locked behind some progression requirement.

The Explorer's role is inspired by Top-Down Action Adventure games (like games on the Legend *of Zelda* series) and the Ghost's gameplay by Tactical games (like games on the *Starcraft* series). The team merged these two game genres in an adventures versus dungeon master like game similar to the dungeon master mode found on *Dungeonland*. This section will highlight the games on these three genres that had a significant influence on the design of *Scare Tactics*.

Legend of Zelda: A Link between Worlds

One of the first games studied was *The Legend of Zelda: A link between worlds* (Figure 4.7). In this game, players take on the role of Link as he explores and defends the kingdom of Hyrule against those that would threaten it. The players have access to a variety of items, each with their unique function, which they must use to get through dungeons and defeat enemies. The core elements of these games are the exploration element and the smart use of both the player's items and the environment when facing obstacles.

Figure 4.7 - The Legend of Zelda: A link between worlds, title screen.

The camera in *A link between worlds* uses a top down perspective (Figure 4.8), but with a twist. The camera is not directly above the character but instead at an angle and zoomed in. This set up not only looks good and allows players to better see details on the character model, but also limits how much of the world players can see on the screen at a time. The game uses the camera distance and angle to limit how much space players have to maneuver in. The size of this "*reaction space*" was carefully designed to not allow players to plan how to deal with enemies as they appear on the screen but still be big enough that players can react to enemies. This keeps players in a constant alert state, waiting for the next enemy to appear on the screen in order to properly react to it.

Figure 4.8 - The Legend of Zelda: A link between worlds, camera.

Monaco: What's Yours is Mine

Monaco: What's yours is Mine (Figure 4.9) is a co-op stealth game where players take on the role of a team of thieves in order to steal all the valuables scattered across the level. Players must choose one of 8 characters to play as. Each character has distinct skills and perks exclusive to them.

Figure 4.9 - Monaco: What's yours is mine, title screen.

Monaco's co-op gameplay relies heavily on the characters in play. Each of the characters distinct abilities complement each other if used wisely. The character composition used by a group of players usually dictates how these players approach levels.

For example,iven a group of three players playing as the *Cleaner,* the *Lookout* and the *Redhead*., the *Cleaner* has the ability to knock out enemy NPCs if they are not alert to his presence, the *Lookout* has the ability to see her surroundings even without line of sight, and the *Redhead* can distract enemy NPCs without harm. With this group of characters the players can go for a more direct approach, using the *Lookout* to scout enemies in order to lure with the *Redhead* and take them out using the *Cleaner*. But if you substitute the *Cleaner* with the *Mole*, who has the ability to dig through certain walls, the strategy changes from taking out enemies to luring them away from the *Mole* while he carves a way to victory. Independent of which strategy the players decide to use, communication between them is essential.

Figure 4.10 shows a level from *Monaco*. Levels in *Monaco* are big and open allowing the players to move around as much as they want. Most rooms on a level are accessible in more than one way, increasing the freedom of movement the players have.

Figure 4.10 - A level from Monaco.

Due to the game's bird's eye view camera, levels are seen from the top. This makes them look like floor plans. This not only fits the game's theme, as thieves often plan their robberies using these kind of plans in movies, games and other media; but it also gives all players information about the level layout and the position of other players. Levels are designed to be neatly separated into sections and this makes communication and planning between players easier. Finally, the game has a "fog of war" on the level. Areas not in the line of sight of players are dark and foggy, allowing players to see only the layout of walls and other objects. This "fog of war" and the lighting effect used in *Monaco*'s levels make them look even better due to the contrast between light and shadow.

Dungeon of the Endless

Dungeon of the Endless (Figure 4.11) is a class-based procedurally generated online multiplayer dungeon crawler game. The game uses handmade rooms on their procedural dungeon generation algorithm. Its algorithm essentially shuffles all the rooms into graphs that fit into a given logic.

Figure 4.11 - Dungeon of the Endless, title screen.

The game also features different character classes that the players must choose. Each class has its own unique stats and skills that complement one another, although not to the extent as the ones in *Monaco*. The main difference from the characters found in *Dungeon of the Endless* when compared to other games is that the plays cannot directly control the characters actions. Players can control to which room the characters go and when to use interactable objects on the environment (research stations), but they cannot control which specific enemy the character attacks. This removes the player from the moment to moment gameplay during a battle, allowing them to think more about their mid- to long-term strategy rather than their immediate necessity.

The co-op aspect of *Dungeon of the Endless* gives the player the option to use different strategies. Individual players can stray away from the main group and explore the map (Figure 4.12). If they are strong enough, they might be able to take on the enemies. More often than not, it gets overwhelming for one player to go solo and stay alive until the end. It also hinders other players as the enemies can come from random directions according to how many rooms have been explored. When exploring a room, players must spend resources to keep said room powered up and prevent enemies from spawning on said rooms. Enemies can destroy the research stations that the players set up (Figure

4.12). Communication and strategy are the keys to success. The later levels of this game can get so complicated that it is impossible to finish them without proper teamwork.

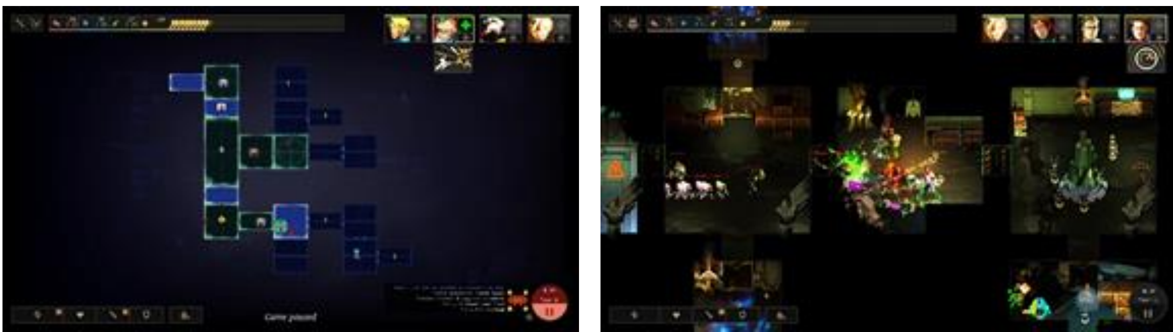

Figure 4.12 - Players in different sections of the map (left) and enemies spawning from several locations (right).

Bastion

Bastion (Figure 4.13) is an action adventure game where the players take on the role of the Kid, a young soldier tasked with guarding the walls of the town of Caelandia. At the beginning of the game the town, and the whole world, is destroyed in an event called the Cataclysm. Being one of the few survivors, the Kid explores this shattered world in order to not only find other survivors but also to fix the caelondian machine called the *Bastion* that can, theoretically, restore the world. This game is famous for its interactive storytelling and narration, however these elements are not the focus of this analysis.

Figure 4.13 - Bastion, title screen.

Bastion's movement and combat control scheme proved to be very interesting when analyzed. The Kid can move in all 8 cardinal directions and has access to a dodge roll. He has access to a variety of weapons, each with two functions and set of special skills; but can only equip two at a time. Movement is controlled by the keyboard while attacking is done with the mouse, when playing using keyboard and mouse controls. Each weapon is attached to a different mouse button and the mouse cursor is used for aiming. When moving, the Kid faces the direction of the movement but, if an attack button is pressed, he will turn to the direction of the mouse cursor while keeping his movement speed and direction (Figure 4.14). This grants players a great amount of maneuverability during combat, since it lets players attack and reposition themselves at the same time.

Figure 4.14 - The Kid aiming his spear at the mouse cursor.

Dungeonland

Dungeonland (Figure 4.15) is a Hybrid Action Adventure/Tactical multiplayer player versus game with a fantasy setting that heavily borrows from common pen and paper RPG tropes, The game takes place in an amusement park called *Dungeonland* created by an evil *Dungeon Master* (the DM) in order to trap and kill adventures dumb enough to come to said park "looking to have some fun". Up to three Players can play on the adventure roles while only one player can play as the DM. This is the game closest to *Scare Tactics* in terms of design out of all analyzed games.

Figure 4.15 - Dungeonland, title screen.

The Adventurer role is divided in three different classes, Warrior, Rogue and mage. Warriors can take a lot of damage and strike slowly with powerful blows; Rogues are fast and can deal weaker blows in rapid succession and Mages are fragile but can use their magic to deal a lot of damage. These classes are designed to be able to stand on their own if necessary but work better with the help of the others. The Adventures objective is to go through the park slaying all monsters the DM throw at them until they defeat a stronger boss monster at the end of each level. They all have a set of 3 unique skills available to them, but players can only use one of them on a given level.

The Dungeon Master role (Figure 4.16) is responsible for placing the obstacles that the Adventurers must overcome. He does that by using a deck of cards that build by the player before the game begins. Cards, once played, can summon monsters or use special skills unique to that card. In order to play a card the DM must spend energy from an energy bar that slowly refills itself. The DM must manage his resources in order to effectively lay down traps and defeat the explorers, before they get to the end of the level and defeat him.

Figure 4.16 - Dungeonland's Dungeon Master mode.

Lessons Learned

After analyzing all of these games and identifying interesting mechanics from them, the design team incorporated these mechanics on their own design in a way that made sense for *Scare Tactics*.

From *Legend of Zelda: A link between Worlds* the camera angle, distance and the "*reaction space*" concept created by them greatly influenced how the *Explorer's* camera was made. The inability to see that far ahead while maintaining enough space to react to obstacles creates intense moment to moment gameplay that was desired for the *Explorer's* gameplay.

The focus on class-based teamwork of the *Explorer*'s role has its roots in the analysis of both *Monaco: What's yours is Mine,* and *Dungeon of the Endless*. The reliance of the classes on one another increases the necessity for good player communication in order to win. Designing the *Explorer* classes in a way similar to that of these games and, therefore, increasing the necessity for player communication, was desired since the team wanted a noticeable contrast between the solo role of the *Ghost* and the team role of the *Explorers.*

The Explorer's movement scheme was based directly from *Bastion*. Early playtests showed that it was easy for the *Ghost* to surround and kill an *Explorer*. The increased maneuverability in the Explorer's movement proved to be necessary in order to create a game that could be balanced and fun, The *Ghost'*s camera was inspired by the one in *Monaco* since it provided a good strategic view of the whole level. This made the *Ghost* player feel more in control of the haunted mansion and allowed him to more easily set up traps and obstacles for the *Explorers*. However being able to see the whole level at once was not possible in *Scare Tactics* simply because the level is too big. The team settled on a more zoomed in view. This zoomed in view had the added benefit of hiding some information from the *Ghost*, adding to the challenge of playing this role.

In *Scare Tactics*, *Minions* are not directly controlled by the player. Once spawned they follow their behavior without any more input from the ghost. This a step further from *Dungeon of the Endless* indirect character control. This decision allowed for the Ghost to focus solely on the strategic placement of *Minions*. As the more tactical of the two roles in *Scare Tactics*, this made sense since it allows the *Ghost* players to focus solely on strategy and not on the moment to moment dangers their *Minions* might face.

Dungeonland, being similar in design to *Scare Tactics*, served as a comparison point. By playing this game it was easy to see the strengths and weaknesses of the design decisions shared by both games. By analyzing *Dungeonland*'s implementation of these design decisions the team was able to come up with alternate solutions better suited for *Scare Tactics*. One such design decision was how the *Minions* should be spawned by the Ghost. By using a mana system and a deck of cards, *Dungeonland* limits the *Dungeon Master* ability to summon monsters in two different ways. This increases the moments of gameplay where "nothing happens", since the DM cannot make anything happen. In *Scare Tactics* the team used a skill and cooldown system instead of the deck of cards, but kept the mana bar. The cooldown served as a weak limitation, stopping the *Ghost* from spamming the same kind of *Minion* constantly, while the mana bar worked as a hard limitation. *Ghost* players are still punished for bad

management of their resources, however these punishments don't happen as often as they do on *Dungeonland* thus reducing the amount of gameplay moments where “nothing happens”.

The study of these other games not only helped the Design of *Scare Tactics* but also increased the design knowledge of the whole team. All games mentioned on this document are impressive in their own unique ways and their study is of great value to anyone interested in game design.

4.4 Level Design

Scare Tactics has evolved from being a top-down 2D game to a 3D game with two distinct camera perspectives. Our initial level was a small flat level and both, *the Explorers and the Ghost*, had a top down camera (Figure 4.17).

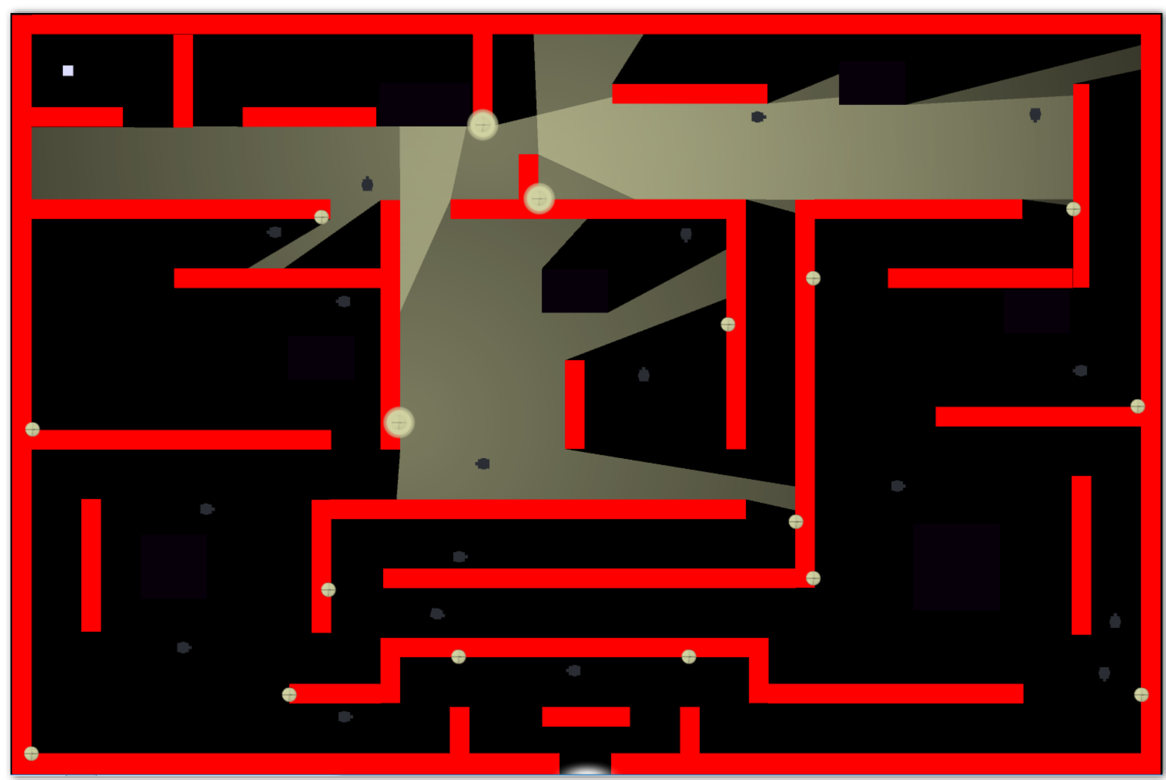

Figure 4.17 - Prototype Level.

Once we chose *Ghost vs Man* as our theme, we needed a new level as it required objectives, which was not present in our prototype level. The player playing as the overwatch took the role of the

Ghost. In our prototype version, the overwatch player used light to their advantage but it did not make sense for the *Ghost*. Thus we had to strategically position the lights again but this time in favor of the players in the scene (Figure 4.18).

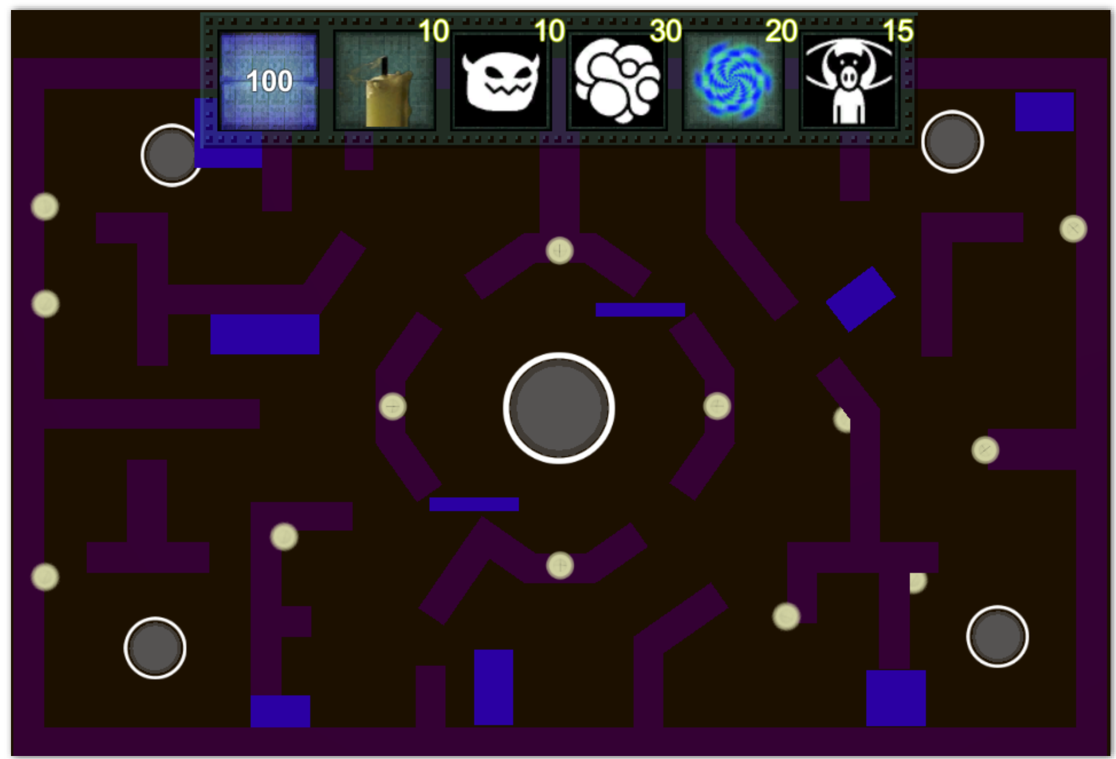

Figure 4.18 - First Level for Ghost vs Man theme.

Since the cameras were different, we wanted to enforce the visual depth and decided to transition from 2D art to 3D art. This was definitely a big transition for us and worked in our favor as it enforced different camera views for both sides of the asymmetrical gameplay. We were happy with the results, but weren't quite satisfied. Even though our art was 3D, our game did not feel 3D because we were still moving on the same flat plane. We wanted to create different height elevations for the players to walk around in. This posed a problem as the grid used for our AI in the game remained 2D. The solution to the problem was to never create areas in the level where an *Explorer* or a minion could walk on two different height elevations. If we were to look at our latest version of the level, the walkable area is still a 2D grid, some parts just have different heights (Figure 4.19).

Figure 4.19 - Scare Tactics Recent Level.

Since the level is a haunted mansion, we looked at mansions around the 1900s for references. We created a list of rooms and areas that we wanted to include in our game. The TV show *Downton Abbey* was a good source of references and helped us get a feel of environment and objects that were used back then. We laid out most of the rooms so it makes sense as a mansion but also kept the game flow in mind. Since the light is used as a mechanic in our game, it was important for us to place the objects and the wall lanterns strategically and not just aesthetically. The objectives for the level were placed to give the *Explorers* a slight advantage since it requires for them to be within a small area. We wanted the final objective to be placed in a bigger area to give a sense of a climax and was thus placed in the main hall of the mansion.

5 Technical Design

This section describes the technical design of *Scare Tactics*. It will discuss the team's choice of tools for the game, the reasoning behind those choices, as well as, the game's overall software architecture.

5.1 Tools

5.1.1 Visual Studio 2015

Scare Tactics is being developed in *C++* using *Microsoft Visual Studio 2015*. One of the team's goals prior to development was to gain more technical experience using *C++*, because of its wide use within the game development industry. As this is ultimately an academic project, this provided an opportunity for the team to tackle challenges unique to *C++* game development, such as memory management and multithreading. *C++* also allows us to have full control over the game loop and submission of draw calls to the graphics API. The graphics API for this project is *DirectX 11*, due to the team's familiarity with it. *Visual Studio 2015* also comes with some useful graphics and GPU debugging tools specifically for *DirectX*.

5.1.2 Unity

In addition, the team used *Unity 5* as a prototyping tool and level editor. This allowed mechanics to be tested faster and without the overhead of *C++* implementation. Once a mechanic was tested and proven, it was ported to the *Scare Tactics* engine. The *Unity 5* scene editor is a powerful tool that allowed the team the luxury of not having to build a level editor. Levels were built in *Unity*, exported to the JSON format, and imported into the *Scare Tactics* engine.

5.1.3 Proprietary Tools (Debugging)

This section quickly describes the proprietary tools developed to increase the development productivity of *Scare Tactics.*

Console Window

We built the console window using *dear IMGUI* framework. It is integrated with our *Trace API*, and it also supports custom commands (Figure 5.1), keyword filtering (Figure 5.2), and copying to the clipboard (Figure 5.3).

```
CONSOLE_COMMAND(trace_test)
{
    TRACE_LOG(1 << " Info messages appear " << "blue" << " in the console..");
    TRACE_WARN("Warn messages appear yellow in the console..");
    TRACE_ERROR("Error: the value is wrong!!! [" << vec3f(1.231f, 0.0f, -123) << "]");
}
```

Figure 5.1 - Usage example of the CONSOLE_COMMAND macro.

```
Warning
[Warning] Skill on cooldown: 1.06895
[Warning] Skill on cooldown: 0.456955
[Warning] Skill on cooldown: 1.17188
[Warning] Skill on cooldown: 0.28125
[Warning] Skill on cooldown: 0.185938
[Warning] Skill on cooldown: 4.26563
```

Figure 5.2 - Console Window filtered by the keyword Warning.

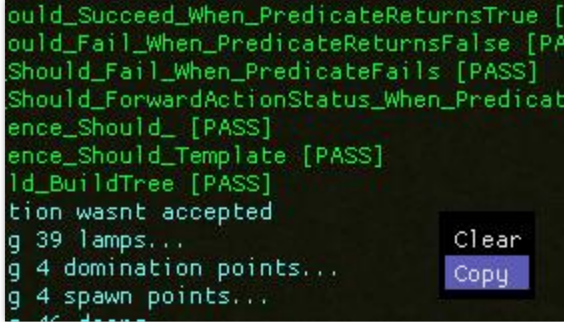

Figure 5.3 - Console Window context menu displaying Clear and Copy to Clipboard commands.

Trace API

In the *Scare Tactics* vocabulary, *Trace* is a powerful set of preprocessor macros that allowed us to inspect and debug our code during development time with zero impact in performance on release mode. *Trace* macros can be subdivided into three main categories: logging, watch window, and line drawing.

Logging

Logs can be created using the *TRACE_LOG*, *TRACE_WARN*, *TRACE_ERROR* and *TRACE* as demonstrated in the (Figure 5.1). The logs are output in the game console (Figure 5.4) as well as in *Visual Studio Output* tab (Figure 5.5).

```
Scare Tactics

$ trace_test
[Log] 1 Info messages appear blue in the console..
[Warning] Warn messages appear yellow in the console..
[Error] Error: the value is wrong!!! [vec3f(1.23, 0.00, -123.00)]
```

Figure 5.4 - Trace messages being displayed in the Console Window.

```
Output
Show output from: Debug
[Log] 1 Info messages appear blue in the console..
[Warning] Warn messages appear yellow in the console..
[Error] Error: the value is wrong!!! [vec3f(1.23, 0.00, -123.00)]
The thread 0x2e134 has exited with code 0 (0x0).
The thread 0x2c678 has exited with code 0 (0x0).
```

Figure 5.5 - Trace messages being displayed on Visual Studio Output Window.

Watch Window

During a debug session, it is common to find key values that need to be watched over time. While it is still possible to observe those values in the console window, they can quickly overflow the console buffer making it impracticable for the developers to follow multiple of those *variable* values in the console output at the same time - and ultimately slowing down the debug process. This scenario is shown exemplified in Figure 5.6.

```
void Level01::VUpdate(double milliseconds)
{
    static double x = 0;
    x += milliseconds / 10000;

    TRACE_LOG("x = " << x);
    TRACE_LOG("Sin(x) = " << sin(x));
}
```

```
[Log] x = 6.22772
[Log] Sin(x) = -0.0554393
[Log] x = 6.22937
[Log] Sin(x) = -0.0537915
[Log] x = 6.23101
[Log] Sin(x) = -0.0521495
[Log] x = 6.23268
[Log] Sin(x) = -0.0504863
[Log] x = 6.23491
```

Figure 5.6 - Variable values being debugged without Trace Window. Usage (left) and output (right).

In order to better address this kind of debugging scenario, the *Scare Tactics Trace API* is loaded with the *TRACE_WATCH* macro. In opposition to the log macros, the watch macro doesn't output to the console window. Instead, it creates a new watch window which displays the watched values in a much more concise way. Figure 5.7 compares the usage and output of the watch macro with the previously shown log macros

```
void Level01::VUpdate(double milliseconds)
{
    static double x = 0;
    x += milliseconds / 10000;

    TRACE_WATCH("x = ", x);
    TRACE_WATCH("Sin(x) = ", sin(x));
}
```

Figure 5.7- Variable values being debugged with Trace Window. Usage (left) and output (right).

Line Drawing

The *Scare Tactics Trace API* is also equipped with several macros for the output of simple lines and geometry to the game screen, as shown in the Figure 5.8. The TRACE_LINE macro is used to draw a straight line segment *from* a determined point *to* another in world space coordinates. The remaining macros draw a combination of lines to form a simple geometry in a predetermined size.

```
#define TRACE_LINE(from, to, color)
#define TRACE_BOX(position, color)
#define TRACE_DIAMOND(position, color)
#define TRACE_CROSS(position, color)
#define TRACE_XCROSS(position, color)
#define TRACE_SMALL_BOX(position, color)
#define TRACE_SMALL_DIAMOND(position, color)
#define TRACE_SMALL_CROSS(position, color)
#define TRACE_SMALL_XCROSS(position, color)
```

Figure 5.8 - Line drawing macro definitions.

The common usage and output of the Line Drawing macros is showed in the Figure 5.9 and Figure 5.10.

```
void Level01::RenderGrid()
{
    for (auto i = 0; i < mAIManager->mGrid.mNumRows; i++) {
        for (auto j = 0; j < mAIManager->mGrid.mNumCols; j++) {
            auto n = mAIManager->mGrid(i, j);
            if (n.weight == -1 && !n.hasLight) continue;

            vec4f color;
            switch ((int)n.weight) {
            case -10: color = Colors::magenta; break;
            case  -2: color = Colors::red;     break;
            case  -1: color = Colors::yellow;  break;
            case   0: color = Colors::green;   break;
            default:
                color = vec4f(0, (30 - n.weight) / 30, .5f)
                    + vec4f((n.weight) / 30, 0, .5f);
                break;
            }

            TRACE_SMALL_BOX(n.worldPos + vec3f(0, 0, -10.6f),
                color * vec4f(1, 1, 1, 0.4f));
        }
    }
}
```

Figure 5.9 - Grid rendering method using TRACE_SMALL_BOX to draw grid data on the game screen. Usage taken from Scare Tactics codebase.

Figure 5.10 - Grid data being displayed with use of a line drawing macro. Example taken from Scare Tactics development build.

5.2 Game Engine Architecture

This section describes research in the area of game engine architecture. One of the goals of this project was to learn about the various subsystems of a game engine, such as rendering, networking, artificial intelligence, memory management and collision detection. Although commercial engines such as *Unity* and *Unreal* can streamline game development, these engines also abstract the subsystems that the team was interested in building. With that in mind, the team decided to build the engine for *Scare Tactics* from the ground up.

The *Scare Tactics* engine is built in *C++ 11*. It currently supports *Windows 10* and *DirectX 11*. It provides a generalized API for rendering, memory allocation, and collision detection. It is

implemented using a combination of *Policy Based Design* and traditional *Inheritance*, which leverages *C++ Templates*.

5.2.1 Components / Policies

In traditional *Inheritance,* a behavior, or policy is embedded in a base class, and then extended via a derived class. This results in class hierarchies with greater depth, which can sometimes be a performance issue when taking in account virtual function calls. Virtual functions, which are typically implemented using a lookup table, often require an extra layer of indirection. (Baggett, 2014)

Policy Based Design, which is also known as the *Component Pattern* (Nystrom, 2014) or the *Strategy Pattern* (Gamma 2011), extracts these policies into classes. This pattern differs from inheritance, because policies are no longer embedded inside of a base class. As Nystrom states, "The entity is reduced to a simple container of components." (Nystrom 2014. 214) This results in class hierarchies with greater breadth.

5.2.2 Template Specialization / Inheritance

C++ Templates are a construct that allows "combinatorial behavior, because they generate code at compile time based on the types (and / or constant values) provided by the user." (Alexandrescu, 2001. 6) Through the use of templates, virtual function overhead can be avoided. By decomposing a class into policies, we can achieve different combinations of behaviors.

For example, a class factory may have different strategies for memory allocation and multi-thread support. A traditional class hierarchy may have one base class and several derived classes that encompass the full suite of desired behaviors, such as single threaded linear allocation or multi-threaded pool allocation. A policy based model builds the factory as a template class that takes an allocation policy and thread policy as template parameters. This allows for greater flexibility as the engine's functionality grows.

5.2.3 Vertex Example

To illustrate how the *Scare Tactics* engine implements policy-based design using templates, this section will describe how *FBX* mesh data is loaded into different types of vertex structures. In graphics programming it is common to define structures to represent vertices of geometric data.

These vertices often contain more than position information, such as vertex normal and texture coordinates used for sampling. The information stored in a vertex structure is determined by the use case. Three dimensional geometry often contains a vertex normal in order to perform lighting calculations, which differs from the vertex used to render two dimensional text.

Managing different types of vertices and the shaders with which they are compatible can often become cumbersome, especially when loading resources from a file. It would be unreasonable to have to modify the *FBX* resource loader for every kind of vertex that will be added. For example, some mesh data is static and contains no animation data, while others are dynamic and contain blend indices and weights to perform vertex skinning. To solve this problem, the *FBX* resource class takes a template argument which is a vertex type. This allows us to add many different vertex types without having to modify the resource class which uses them. The *FBX* resource class can load models using a vertex that stores only positions, or a vertex that stores positions, normals, texture coordinates and blending information.

5.3 Class Breakdown

The domain model in *Scare Tactics* is a combination of traditional *inheritance* and the *component* pattern as described in section 5.2.

5.3.1 BaseSceneObject

The *BaseSceneObject* is the base class for all other non-component entities within the game. It is an instantiable class, which is composed of an instance of a *Transform*. A *Transform* is a class responsible for maintaining the position, orientation, and scale of an object. Any entities which require a physical

representation in the game will have a *Transform* and thus inherit from *BaseSceneObject*. For example, the *Explorer and Minion* classes inherit from *BaseSceneObject*. Figure 5.11 illustrates the *BaseSceneObject* diagram.

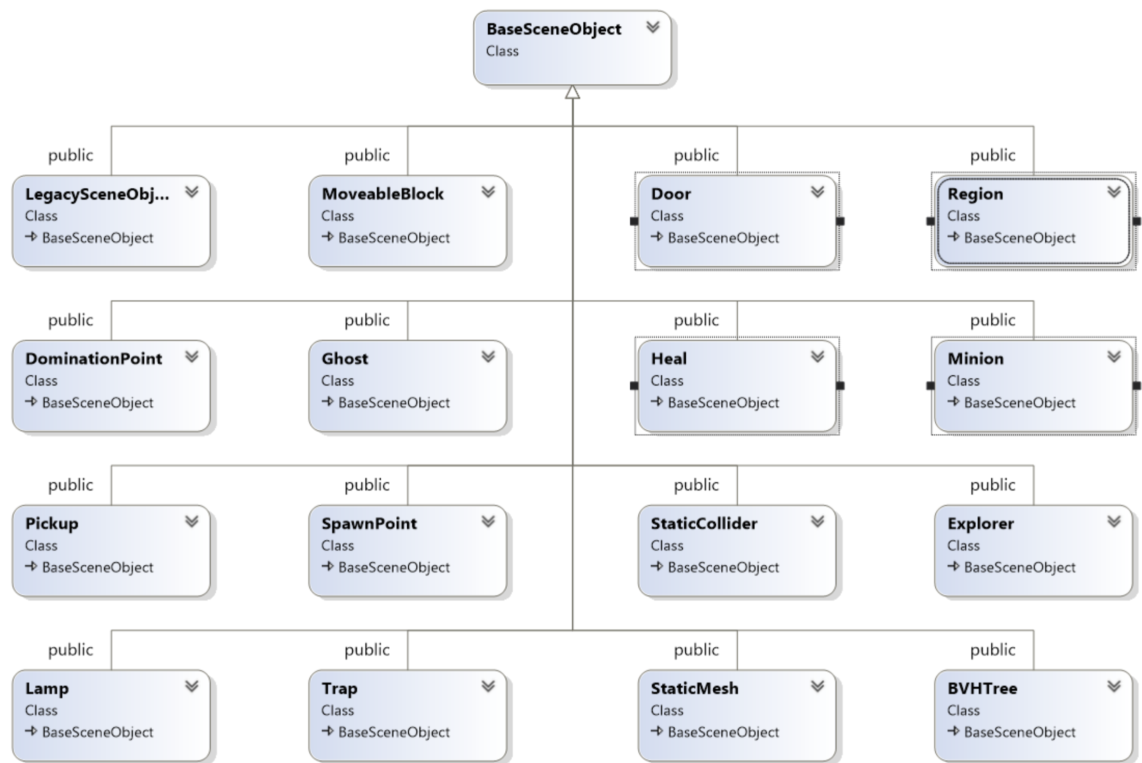

Figure 5.11 - BaseSceneObject Class Diagram.

5.3.2 BaseComponent

The *BaseComponent* class is the base class for all component classes. Any class which represents a behavior or policy will inherit from the *BaseComponent* class. The purpose of this class is to be attached to classes derived from *BaseSceneObject*. It maintains an active state, and a pointer back to the owning instance of *BaseSceneObject*. It also provides facilities for exposing callback functions for a derived *BaseComponent* class. Figure 5.12 illustrates the *BaseComponent* diagram.

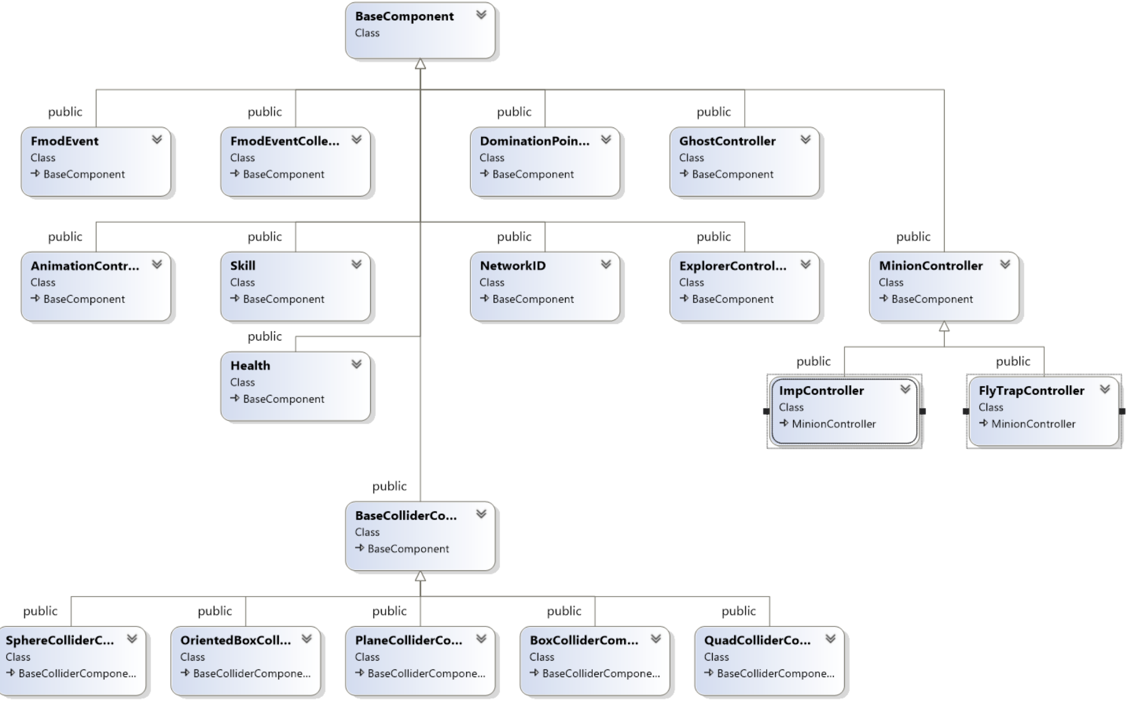

Figure 5.12 - BaseComponent Class Diagram.

Some examples of derived component classes are the Health, and ColliderComponent classes. The Health class is responsible for keeping track of the owning *BaseSceneObject* instance's health and receiving callback function's when that value is synced across the network.

The ColliderComponent class is responsible for defining a volume with which to test for overlaps and notifying the owning *BaseSceneObject* instance when collisions have occurred and concluded.

5.3.3 Explorer Example

As an example that illustrates how this design works, one can look at the *Explorer* class. The *Explorer* class is derived from *BaseSceneObject*. It contains the following component classes: *NetworkID*, *ExplorerController*, *AnimationController*, *SphereColliderComponent*, *Skill*, and *Health*. Each component has a single responsibility and exists across multiple entities. Figure 5.13 illustrates the Explorer class and its fields.

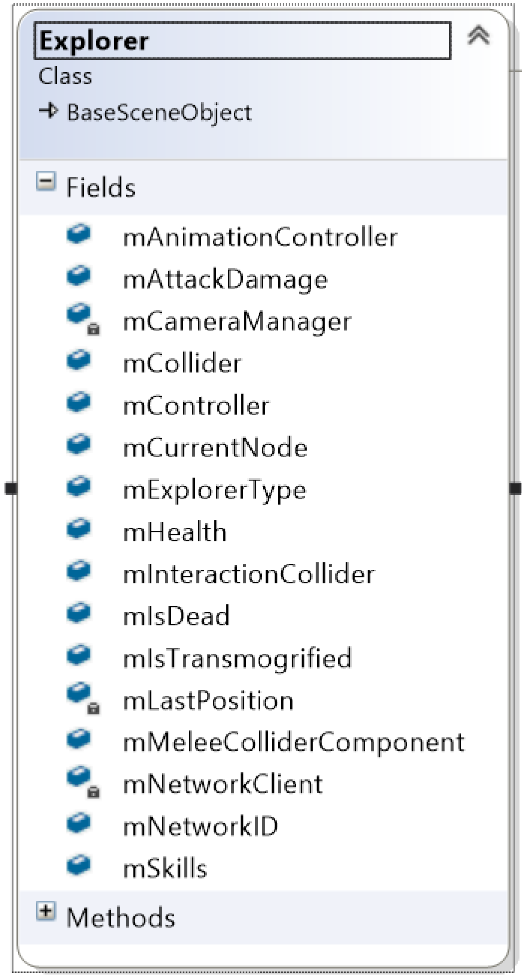

Figure 5.13 - Explorer Class Diagram.

The *NetworkID* class is responsible for identifying the *Explorer* to a network server or client. The *ExplorerController* is responsible to handling game logic of the *Explorer*. It handles callbacks from other components such as the *AnimationController*. The *AnimationController* is responsible for managing the animation state of the *Explorer*. It stores the *SkeletalHierarchy* and animations for a given model. The *SphereColliderComponent* class stores the bounding sphere of the model and is used for intersection testing with static scenery such as walls and floors. The *Skill* class manages the activation and cooldown times of the different abilities of the *Explorer*. Lastly, the *Health* component manages the maximum, and current health values of the *Explorer* and provides an interface to manipulate that data.

5.3.4 Object Factory

The *Object Factory* is among the fundamental pieces of the *Data Driven* design adopted during the development of *Scare Tactics*. It was built on top of our *Pool Allocator* and it also supports C++11 style iterators (Figure 5.14).

```
for (auto& d : Factory<DominationPoint>())
{
    d.mController->mProgress = 0;
    d.mController->isDominated = false;
    for (int i = 0; i < MAX_EXPLORERS; i++)
        d.mController->mExplorers[i] = nullptr;
}
```

Figure 5.14 - Usage of Factory iterator.

The *factory* requires all object classes to be registered prior to its usage. This registration is made during precompile time using the *REGISTER_FACTORY* macro (Figure 5.15).

```
// register SCENE OBJECTS factories
REGISTER_FACTORY ( Ghost,           1    )
REGISTER_FACTORY ( Explorer,        MAX_EXPLORERS)
REGISTER_FACTORY ( Minion,          MAX_MINIONS )
REGISTER_FACTORY ( StaticMesh,      MAX_STATIC_MESHES)
REGISTER_FACTORY ( StaticCollider,  MAX_STATIC_COLLIDERS)
REGISTER_FACTORY ( SpawnPoint,      5    )
REGISTER_FACTORY ( DominationPoint, 6    )
REGISTER_FACTORY ( Lamp,            MAX_LAMPS)
REGISTER_FACTORY ( Region,          100)
REGISTER_FACTORY ( Door,            MAX_DOORS)
```

Figure 5.15 - Object registration.

Figure 5.16 illustrates how objects are created while the Figure 5.17 illustrates how objects are destroyed.

```
mNetworkID = Factory<NetworkID>::Create();
mNetworkID->mSceneObject = this;
mNetworkID->mHasAuthority = mNetworkManager->mMode == NetworkManager::SERVER;
mNetworkID->RegisterInteractCallback(&OnInteract);
```

Figure 5.16 - Object creation.

```
Lamp::~Lamp()
{
    Factory<OrientedBoxColliderComponent>::Destroy(mTrigger);
    Factory<NetworkID>::Destroy(mNetworkID);
}
```

Figure 5.17 - Object destruction.

5.4 Graphics

This section describes the rendering structures, and techniques used in *Scare Tactics*. *Scare Tactics* presented some interesting rendering challenges based on the requirements of a scene and the coupling of the lighting and artificial intelligence systems. The game features a full 3D environment, animated characters and supports shadow casting light sources. Render passes are performed in the following order: shadow textures, static mesh data, animated mesh data, light volumes, and the final composite pass.

5.4.1 Shadow Mapping

Shadow Mapping is a projective texturing technique where a texture is created that represents the scene from the perspective of the light. This texture carries depth information of the scene and is used during the light buffer pass. In this pass, we compare the distance of the pixel from the light source to the depth stored in the shadow map. If the distance is greater, than it can be determined that the pixel is not in a direct line of sight of the light source (Luna, 2012. 673).

5.4.2 Hardware Instancing

Game scenes are constructed using modular walls and floor models. There are also a number of static objects that serve as scenery. The renderer makes heavy use of *Hardware Instancing* in this pass.

Instancing is a technique where a geometry data is rendered multiple times using different transform data (Gregory, 2009. 850). In order to minimize the setting of GPU state, transform data is grouped by mesh and passed to the GPU in an *Instance Buffer*.

5.4.3 Lighting Deferred Rendering (Point lights / Spot lights)

The lighting system in *Scare Tactics*, is required to support anywhere between 10 - 20 shadow casting light sources per scene. Given this constraint, *Scare Tactics* uses a *Deferred Lighting* system with *Shadow Mapping*.

Deferred Lighting is a post rendering process, where various information about a scene is rendered to several textures. This collection of textures is called a *Geometry Buffer* (or *G-Buffer*), and normally contains textures for position, surface normals, diffuse color and depth. Leveraging this information, another pass is performed using the geometry of a light volume to calculate the lighting information of the scene. This is sometimes called the light buffer texture. The light volume for a point light is a sphere and for a spot light it is a cone. The final pass is performed which uses the information stored in the *G-Buffer* and light buffer texture to compose the final image. The reason this method is preferred for scenes with many lights is because via rasterization the only pixels drawn are pixels that are contained within the lights volume. This contrasts forward lighting where light buffer is calculated in the same pass as geometry, which will perform lighting calculations for pixels that will not be in the light's volume. An example of G-Buffer textures can be viewed in **Figure 3.4.1**.

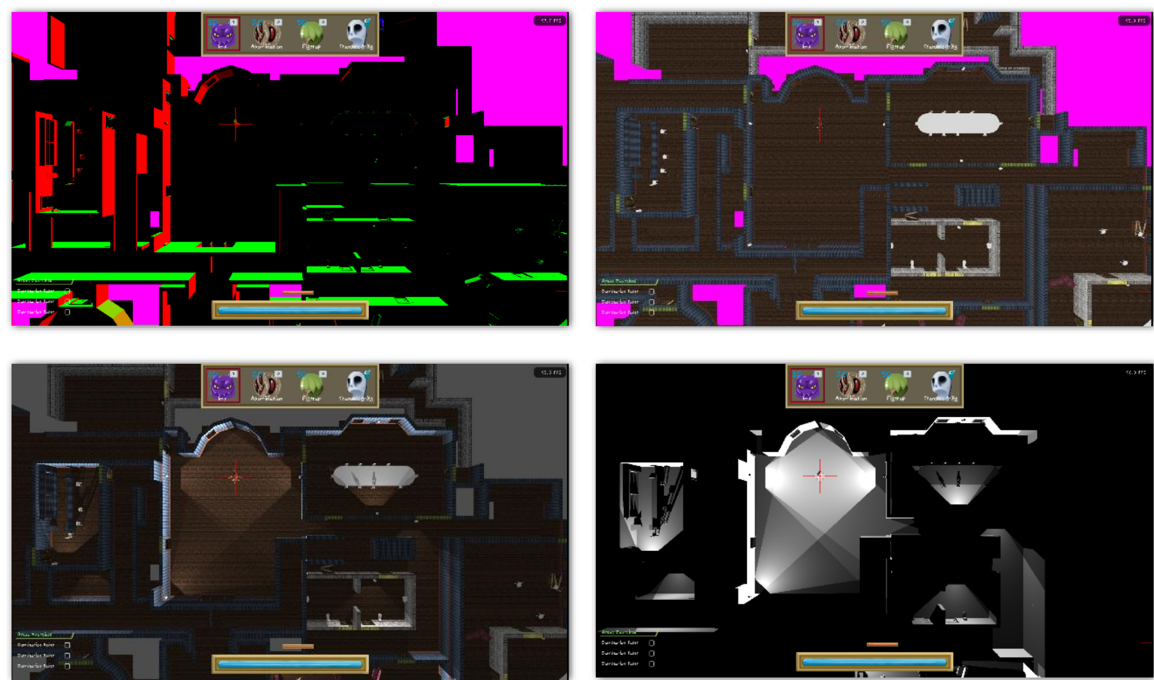

Figure 5.18 - G-Buffer is composed of several textures, including a normal texture (top left). Diffuse texture (top right). A lighting texture (bottom right) is created using the normals and positions. The final image (bottom left) is composed using the diffuse and lighting textures.

Light also serves the purpose of inhibiting the *Ghost's* abilities. For example, The *Ghost* cannot spawn enemies for the *Explorer's* inside of well-lit areas. Also the enemies that are spawned become slower when traveling through well-lit areas, and will try to avoid those areas. This requires that the lighting information of a scene somehow be translated into a format which can be used by an AI to plan motion and behavior. Light buffer textures are also used by the artificial intelligence system to update grid information for enemy pathfinding.

5.5 Skeletal Animation

In *Scare Tactics*, all 3D characters are animated using a technique called *Skeletal Animation.* In this technique, a hierarchy of coordinate spaces called *bones*, or *joints* is constructed. This hierarchy is referred to as a *skeleton*. Each vertex of a mesh is "skinned" to this skeleton, which means that each vertex is influenced by a subset of the skeletons joints. An animation is a series of keyframe poses of

the skeleton. *Keyframes* are position and orientation information of each joint at a given time during an animation. As an animation plays, the positions and orientations of the joints are interpolated and used to morph the position of the vertices over which they have influence. (Gregory, 2009. 547)

The characters in *Scare Tactics*, are imported via *FBX* files, have a max joint count of 64 and max influence count of 4 joints per vertex. Animations and joint matrices are calculated on the CPU and then passed in a *Constant Buffer* to the GPU. On the GPU each vertex is transformed by a weighted average of joint matrices before rasterization.

5.5.1 State Based Animations

Sitting above the low level implementation is an *Animation Controller*, which supports *State Based Animation.* Each character has an idle, walk, attack state. Each state is a mapping between a range of animation keyframes to a behavior. From a gameplay perspective, switching between animations is easily accomplished by switching the state of an animation controller. The animation controller also provides the ability to tag a specific keyframe with a callback function. This allows the user to time behavior with a given animation. For example, melee animations often have a function tagged to the beginning and end of the animation in order to activate and deactivate a collider used for hit detection.

5.6 User Interface

This sections describes some of the considerations for the implementation of the *user interface (UI)* of *Scare Tactics*. Since the initial prototypes, we quickly noticed how important UI would be to convey most of the game's mechanics, goals and controls. During the different development stages, we tested different combinations with different groups of people in order to learn what worked best in terms of interface design. Soon enough, we noticed our UI was becoming complex and dense in terms of meaning - becoming one of the key aspects of the game.

Given the complexity of the prototype UI, one of the first solutions we considered was to use a fully-fledged HTML engine to render the elements into some texture and then use that in our graphics

pipeline (e.g. *Awesomium*). This solution, despite being easy and quicker than any alternative, did not met two of our goals: it is not performatic and did not add much in terms of knowledge, being a black-box solution.

The solution we finally used involved creating a custom *Sprite Manager* to render all of our 2D elements - both for static *HUD* elements (e.g. skill bar) and for world-based elements (e.g. health bars). We created a simple interface/API that allowed programmers to use the sprite manager flexibly from different parts of the code for different purposes, yet, everything was rendered with a single draw call using hardware instancing. The shader handling sprites is also responsible for some effects, such as fill color and direction. The Sprite Manager also offers auxiliary methods to position UI elements both on world space and camera space.

The following images shows some of our *Sprite Manager* capabilities: icon/text alignment (Figure 5.19), linear fill (Figure 5.20), radial fill (Figure 5.21) and world space rendering (Figure 5.22).

Figure 5.19 - Different UI elements on different alignments.

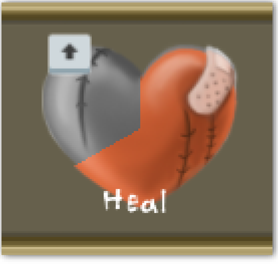

Figure 5.20 - Radial fill being used to indicate the cooldown of an ability. The math is part of the shader that render sprites.

Figure 5.21 - Linear fill being used for Ghost's mana bar.

Figure 5.22 - UI rendering based on world space coordinates.

5.6.1 Font rendering

While dealing with user interface, one problem that we had to deal with was font rendering. Different solutions exist, most of them involving the creation of font maps of different sizes. We decided to implement the signed distance field (SDF) solution (Green, 2007). This solution allowed us to use the same font map for different scales, drastically reducing the number of textures to be loaded to the GPU.

5.6.2 Debug UI

During early development, several of our debug tool and solutions (e.g. console, FPS meter, BVH explorer) were created and used a third-party library for the user interface, *dear IMGUI*. By using this library early on, we postponed the development of our actual UI solution to a later phase, without hindering other areas. More importantly, some of the design solutions from *dear IMGUI* were used as guidelines when creating ours, especially the *Immediate Mode* concept, opposed to more commonly used *Retained Mode* paradigm. Some of these tools are further detailed in section 5.1.3.

5.7 Collision Detection

Collision Detection is the process of determining whether two or more objects are overlapping. (Ericson, 2005) It is a broad topic with many applications and techniques. This section describes the implementation of collision detection in *Scare Tactics*. Figure 5.23 shows an overall diagram of the collision detection architecture.

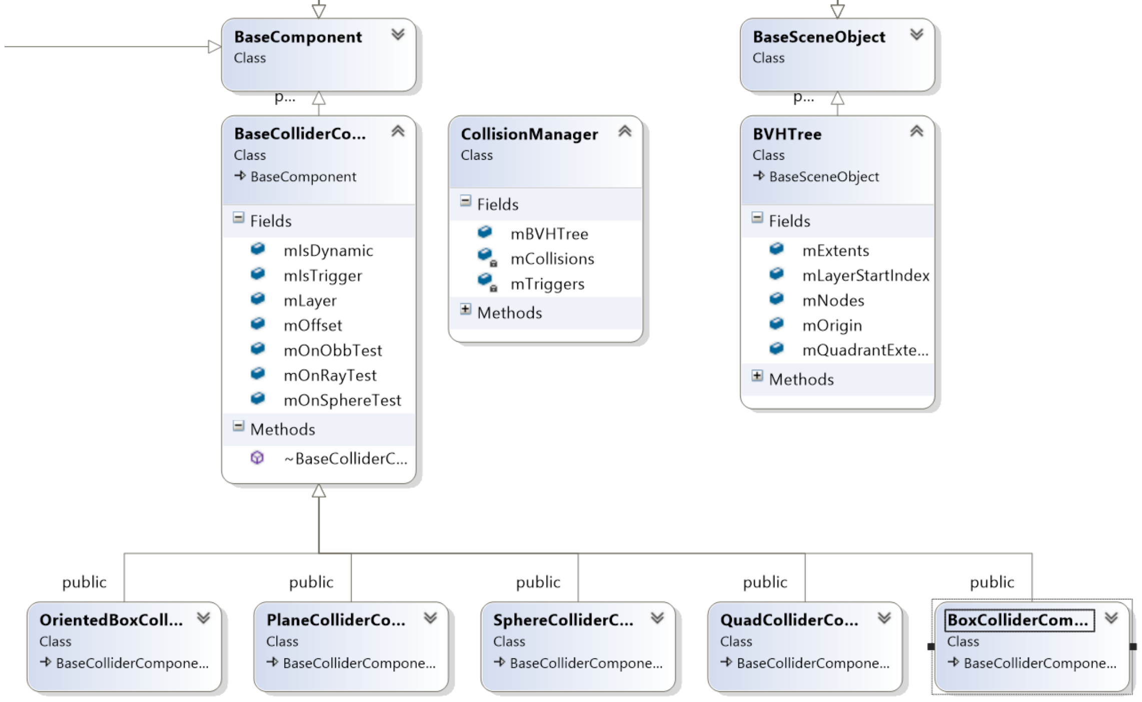

Figure 5.23 - Class diagram for collision detection system.

5.7.1 Template Colliders

The *Scare Tactics* engine collision detection is composed of several layers (Figure 5.7.1). The first layer is a collection of templated structures used to represent parametric surfaces and objects, such as rays, lines, planes, spheres, and cubes. Also included in this layer are basic intersection tests for these objects, such as the intersection tests for a ray and plane or a sphere and cube, etc. Many of these tests are based on iterations over each dimension of the object. For example, the intersection test for a ray and cube is basically a test between a ray and each pair of parallel plane faces of the cube. In 2D, this test iterates twice. Once for each pair of faces parallel to a given basis axis. In 3D this test iterates thrice. Our parametric structures and intersection tests use *C++ Templates* to generate code specific to testing in 2D or 3D.

5.7.2 Collider Components

The second layer is a collection of component classes that wrap an instance of a first layer class. For example, a sphere collider component wraps an instance of a sphere struct that takes a 3D vector template argument. The component layer is responsible to storing information about the collider, such as the active and dynamic state. This information determines whether collisions will be tested against this collider and if that test will be a dynamic or static collision test. This layer also discerns whether the collider should be treated as a physical object or as a trigger for some type of user defined behavior.

A collider that is not a trigger is treated as a physical object and overlaps will be handled automatically by the collision engine. The user does not have to include logic for objects that simply must not overlap. A trigger will not be processed as a collision and simply notifies the overlapping objects of each other’s presence.

5.7.3 Bounding Volume Hierarchy

The last layer is composed of a *Bounding Volume Hierarchy* and a *Collision Detection Manager*. These higher level objects work in unison to process collisions and dispatch events. A *Bounding Volume*

Hierarchy is a tree of nodes representing objects that contain smaller objects. This is a spatial partitioning technique to optimize the process of finding collisions by discarding large groups of objects. (Ericson, 2005. 236)

In the *Scare Tactics* engine, after the scene is loaded, it is broken up into quadrants, which are the largest volume in the scene. Objects in the scene are then process in decreasing size order and added to the hierarchy. Dynamic objects, such as characters, are removed and added to the bounding volume hierarchy each frame. When an object is added, it is first tested against a quadrant. Once finding the intersecting quadrant, all objects contained in nonintersecting quadrants are discarded, which results in faster iteration times. Figure 5.24 shows how the *Bounding Volume Hierarchy* tree is organized for one screen.

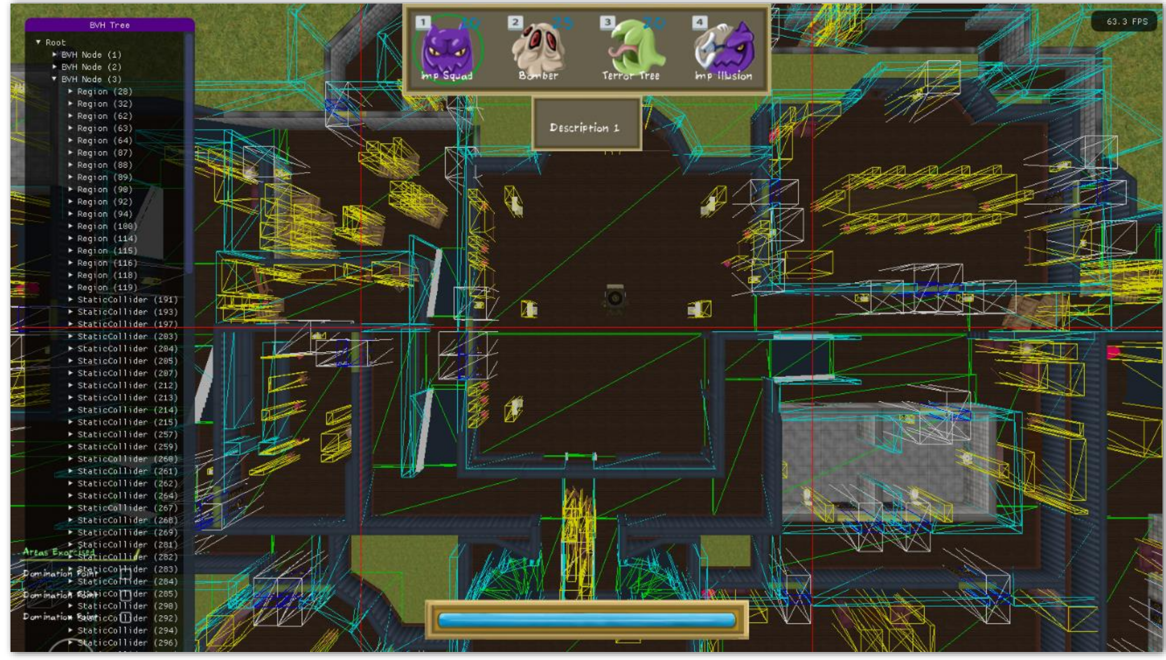

Figure 5.24 - Bounding Volume Hierarchy. Root Quadrants are traced in red. All other colliders are a child of one or more quadrants.

5.7.4 Culling

Apart from these previous layers is a culling layer. This layer is application specific and used during non-instanced rendering passes. For example, when rendering shadow textures, objects are culled

against the light's perspective frustum. This is performed to avoid processing geometry on the GPU that will get discarded during rasterization.

5.8 Networking

This section describes another important research field for *Scare Tactics*: *networking*. Given the asymmetric gameplay we were aiming for, with distinct play styles and hidden information, it was important to ensure that players could experience our game on different machines. Therefore, we researched different techniques to implement multi-player networking in our game. Initially, *Unity* helped to sketch and test some of the most basic decisions in terms of the network architecture to use in our final game.

One interesting point is that we have settled on a "one versus many" experience. This decision immediately allowed us to draft a network architecture in which the *Ghost* player is the host environment and the multiple *Explorers* are clients. This design has been put to test and allowed us to set up a playable prototype very quickly. Most of the mechanics involved a handful of transform synchronizations and some still required more reliable commands, but both scenarios were easily handled by the new networking module in *Unity 5*.

The greater challenge came with the *C++* implementation. We decided to try and develop our own network module, without resorting to third-party libraries. This decision was in line with the academic goals we set for our project. Moreover, this gave us a lot of control over packet creation and management, allowing us to have a very simplistic yet robust network module behind our game. Also, as we will detail further in section 5.10.2, we could optimize the package serialization/deserialization to ensure some extra performance.

The *C++* implementation mimics some of the design decisions used in the *Unity* prototype - most notably the unified pattern in which one component can be present on both a client and a host, having different behaviors associated with it. Also, every network component has the concept of

authority, guiding who is ultimately responsible for a certain object. We have also used the general differentiation between a *Command*, i.e., a function that a client can call on the host given it has authority over the target object, and a *RPC (Remote Procedure Call)*, i.e., a function the host can call on one or more clients to update certain information.

We used basic *socket* functions made available by the *Windows API* to implement both host and client, using a customized protocol transmitted through *TCP*. We chose *TCP* over other alternatives, noticeably *UDP*, because it allowed us to simplify gameplay implementation. *TCP* ensures packet delivery and sequence, so that our game does not have to worry about scenarios in which packets have been lost. This comes at a cost, especially in terms of packet overhead. However, after some tests, we could notice that using asynchronous sockets and disabling *Nagle*'s algorithm on them was enough to obtain reasonable performance.

Initially, our game has no goals to be played over the Internet, so our design and optimizations focused only playing via LAN connections. Also, for simplicity and given our current scope, the connection is plain and not encrypted or secured. This helped us during development and debugging, as we could read the protocol and inject packets when necessary. Should our game evolve to a commercial product, these decisions would need to be reevaluated.

Connections are handled by a *NetworkManager* class, which is a singleton in our solution. This class exists on the host and the clients, and is aware of their roles. Therefore, our main update loop can address networking updates without worrying about any special treatment.

5.9 Artificial Intelligence

This section describes the use of *Artificial Intelligence* in *Scare Tactics*. In particular, it describes the *motion planning* and *decision making* techniques used to mold the behaviors of the different *minions* that compose *Scare Tactics*.

5.9.1 Motion Planning

Motion Planning is the process of breaking down a desired movement sequence into a set of steps that satisfy the movement requirements. In *Scare Tactics*, *Motion Planning* gives *minions* the ability to chase *explorers* or wander around a certain area of the level.

Choosing a suitable pathfinding strategy was one of the challenges faced during the development of *Scare Tactics*. It involved figuring out an optimized way to guide several NPCs into their respective targets. Initially, we tackled this problem with a simplistic solution: each AI agent would perform a new A* query from its position to its target position. Additionally, the A* would be recalculated whenever the target position the layout of the grid changed. Soon, this solution proved itself as a bottleneck in the scalability of the game and we had to look for optimization strategies to the pathfinding solution.

After analyzing the possibilities of improvement on the pathfinding algorithm, such as Jump Point Search (Harabor, 2011), Hierarchical Pathfinding (Millington, 2009, 262) and D* (Millington, 2009, 272) - we decided to implement a variation of wave-front expansion using GPU resources to speed up the process (Durant, 2013) (Cossell, 2011, 191). The final solution uses a two-dimensional grid as the search space and is composed by the following steps:

1. The grid data is initialized on the CPU. At this point, each node on the grid holds its position, its 2D coordinates on the grid and weight of -10 (this is an arbitrary value that indicates that the node hasn't yet been processed by the GPU). Then, the positions of all *Explorers* are marked in the grid by changing the respective node weight to 0.
2. The grid data is then copied to a shader resource in the GPU for further processing.
3. On the GPU side, GPU step 1 processes the information from the actual level, e.g. lit or unlit areas and wall placements, by sampling from textures previously used for rendering purposes. This is reflected on different weights on the grid.

4. GPU step 2 iterates multiple times through the grid data starting off from the *Explorer* nodes. For each pass, it performs a wave-front expansion. Here is where we limit the range of the *minions*, for the number of passes reflects to which extent an *Explorer* is within the "sight" of the AI agents.
5. The grid data is then copied back to the CPU with the updated weight values.

The updated grid works as a distance field. Node weights can either carry a negative value indicating that the node is not walkable, a zero value indicating that the node is a goal node, or a positive value indicating the number of steps that the node is from the closest *Explorer*. With this solution, it's not necessary to find the whole path from an AI agent to the closest goal. Instead, AI agents can simply walk towards the neighbor node with the smallest positive weight until a goal node is reached.

5.9.2 Decision Making

The success of high-profile games like Halo 2 (Bungie Software, 2004) has leveraged the popularization of *Behavior Trees* over the last decade. They emerged as a scalable alternative to the popular *Hierarchical Finite State Machines* (*HFSM*).

In *Finite State Machines*, states, the *FSM* and *HFSM* building blocks, hold a reference to the next *state* to be executed. These references are called *state transitions* and are commonly manifested as a simple call to a *set-state* method pointing to the next-to-be-executed *state*. *States* provide developers with a simple way of encapsulating the code for different behaviors into specialized modules with some hardcoded *transitions* (Nystrom, 2014, 100).

The building blocks of *Behavior Trees*, often called *tasks* or *behaviors*, do not hold a reference to the following *task* to be executed. In other words, a *task* does not declare explicitly by the means of a *task transition,* which other *task* should execute next. Instead, they are added to a parent scope, and executed according to the semantics of the parent scope. This self-containing factor combined with

some parametrization (Millington, 2009, 334) allows the structure of the *Behavior Tree* to be easily rearranged, and its *Tasks* to be easily used in different contexts (Champandard, 2007).

Behavior Tree tasks are split into four major types: *Actions*, *Conditions*, *and Composite* and *Decorator tasks*. *Actions* and conditions are the leaf *tasks* on the tree. These *tasks* hold game specific logic while composite and decorator *tasks* define the structure and execution flow of the tree. Such *tasks* should be able to execute context free. The power of *Behavior Trees* lies in the different ways in which these *tasks* can be mixed and matched (Millington, 2009, 335).

Action tasks are the main building block for *Behavior Trees*. Conceptually, each *action* should perform a small chunk of the objectives to be performed as a whole. For example, in order to complete its objectives, an AI agent must open doors, turn on and off lights, and walk towards different target locations. In this scenario, there could be an *Action* to walk towards a set target, another *Action* just to set the target to a specific entity or location, and a third *Action* to interact with an object (a light switch to be turned on or off, or a door to be opened or closed).

Conditions check the state of the game. In the previous example, there could be *Conditions* for checking the proximity between the AI agent and its target, checking the state of doors and lamps, checking the internal state of the AI itself (do I have a valid target? Do I have enough life points?), and so on. In *Scare Tactics*, condition *tasks* are called *Predicates*.

Composite and *Decorator tasks* act like the branches of the tree. Typically, *Composites* handle multiple children while decorators act like a wrap around a single child (or, in specific cases, a predefined number of children) (Champandard, 2007).

There are two fundamental types of *Composite tasks*: sequences and selectors. “Both of these run each of their child *tasks* in turn”, “a *Selector* will return immediately with a failure status code when one of its children fails” and “a *Selector* will return immediately with a success status code when one of its children runs successfully” (Millington, 2009, 335). In other words, *Sequences* are responsible for executing each of its children one by one but will break the execution as soon as one

of the children fails to execute, and *Selectors* will run its children one by one until one of them executes with success.

Decorator tasks can be used to alter the standard behavior of other *tasks* without modifying the contents of the referenced *tasks*. That is done by plugging the *Decorator tasks* in between the original *tasks* and its parent *tasks* (Figure 5.25). A classic example of *Decorator tasks* is the *Repeat task*, which executes its child a predetermined number of times before returning with success to its parent.

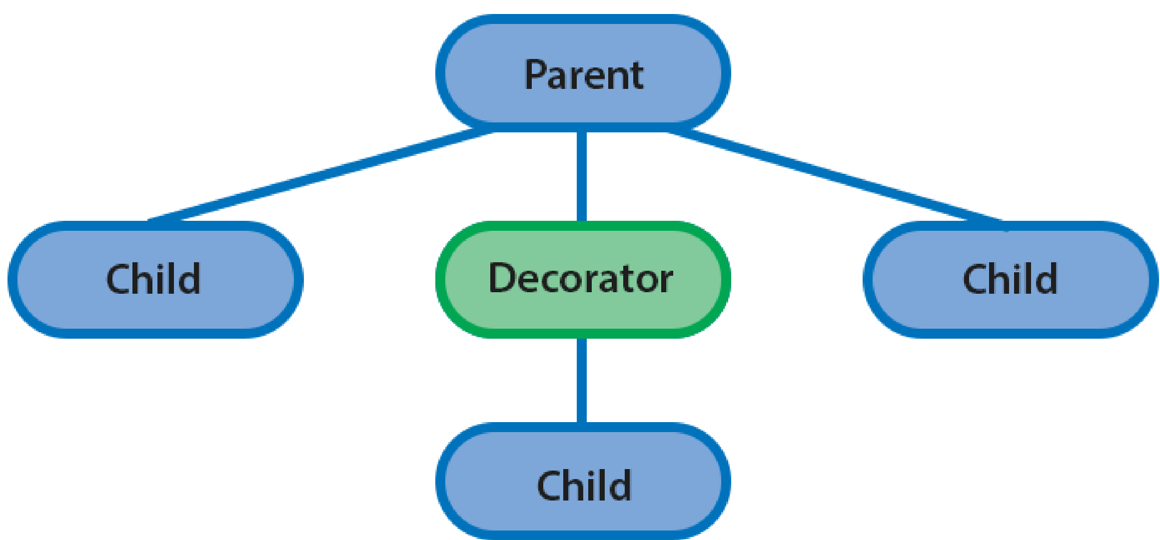

Figure 5.25 - Typical structure of a decorator task.

Behavior Trees are used in *Scare Tactics* to control the different minions present in the game. We used the concept of *subtrees* to maximize the reusability of the behavior trees (Figure 5.26 and Figure 5.27). On the outside, subtrees work just like any other task and can be attached normally to any composite or decorator node. But under the hood, each *subtree* creates a new localized context that works autonomously.

```
Tree& MinionController::CreateAttackSubtree()
{
    return TreeBuilder(mAllocator, "(-->) Attack")
        .Composite<Parallel>()
            .Decorator<Mute>()
                .Conditional()
                    .Predicate(&IsExplorerInAttackRange, "(?) Is Explorer in Attack Range")
                    .Action(&StartAttack, "(!) Start Attack")
                .End()
            .End()
            .Conditional()
                .Predicate(&IsAttackInProgress, "(?) Is Attack in Progress")
                .Composite<Parallel>()
                    .Action(&TargetClosestExplorer, "(!) Target Explorer")
                    .Action(&LookAtTarget, "(!) Look at Target")
                .End()
            .End()
        .End()
    .End();
}
```

Figure 5.26 - Subtree declaration pulled from Scare Tactics codebase.

```
ImpController::ImpController()
    : MinionController()
{
    mBehaviorTree = &TreeBuilder(mAllocator)
        .Composite<Priority>("(/!\\) Priority Selector")
            .Subtree(CreateKnockbackSubtree())
            .Subtree(CreateAttackSubtree())
            .Subtree(CreateChaseSubtree())
            .Subtree(CreateWanderSubtree())
        .End()
    .End();
}
```

Figure 5.27 - Imp Behavior Tree assembled from various subtrees. Example pulled from Scare Tactics codebase.

5.10 Optimizations

In preparation for the development of the *Scare Tactics*, several prototypes were built. The first prototype, which was built in *Unity* suffered from slow performance due to the amount of collision tests and AI routines being performed. This served as a motivation to build the project in C++, which would allow us more control over the various game engine systems. One of the benefits of building

the game's engine were the opportunities to optimize based the specific game being built. Two areas of note where this occurred where in memory management and networking.

5.10.1 Memory Management / Static vs Dynamic

The goal of the memory management system is to allow developers to allocate memory efficiently without having to perform data alignment and pointer arithmetic. Several types of allocators were built for various contexts, such as a linear allocator. This type of allocator cannot free individual allocations but rather frees all allocations at once. It is useful for allocations that will not take place often. For example, linear allocators are used for data that is created once per level such as textures and meshes.

Another type of allocator the engine uses is a pool allocator. A pool allocator is useful for rapid allocations of objects of the same size. It works by allocating a large block of memory and then splitting that memory into smaller chunks. Internally it behaves as a singly linked list of memory locations. The task scheduling system in the engine uses a pool allocator for task storage.

The motivation behind efficient memory allocation is performance. Performance is largely governed by memory access patterns of modern systems architecture. Most modern architectures feature a cache system where data can be retrieved faster than from RAM, as shown in Figure 5.28. A cache hit occurs when the CPU retrieves data from the cache. Program optimization accomplished in part by maximizing cache hits. Memory allocation plays a part in this because when data is read into the cache, neighboring data is read as well. For example, a set of data that needs to be transformed in some fashion that is spread out in memory will result is more cache misses than data that is allocated contiguously.

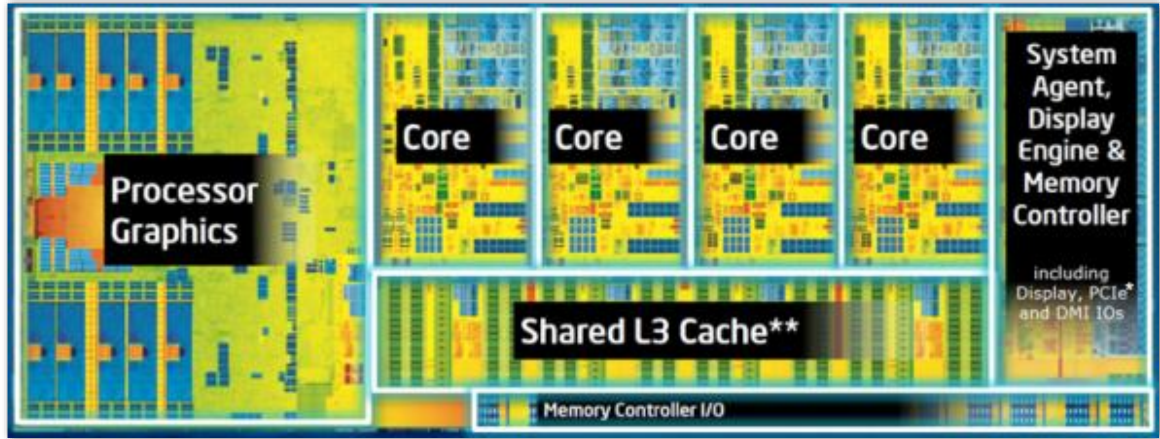

Figure 5.28 - Intel CPU architecture.

5.10.2 Packet Construction

By looking at the communication requirements of *Scare Tactics,* we could sketch a factory for network packets that was simple enough to cover all packets we would need to run our game. This factory is based on the simple concept that every packet we transmit has a fixed size, yet, according to the initial byte, it is possible to deal with it differently. Figure 5.29 illustrates the different packets we use and how they use the same amount of memory regardless of their contents.

```
enum PacketTypes {
    INIT_CONNECTION = '0',
    SET_CLIENT_ID   = '1',
    SPAWN_EXPLORER  = 'E',
    SPAWN_SKILL     = 'S',
    GRANT_AUTHORITY = 'A',
    SYNC_TRANSFORM  = 'T',
    SYNC_HEALTH     = 'H',
    SYNC_ANIMATION  = 'N',
    INTERACT        = 'I',
    READY           = 'R',
    RESTART         = 'r',
    DISCONNECT      = 'D',
    UNKNOWN
};
```

Figure 5.29 - Different packet types.

This solution allows us to have almost zero processing time to serialize and deserialize the data coming through the network. A simple memory copy operation can be used to populate the packet structure (Figure 5.30), and use it according to the scope in which the packet is required.

```
struct Packet {
    PacketTypes Type = UNKNOWN;

    union {
        unsigned int UUID = 0;
    };

    union {
        unsigned int ClientID = 0;
    };

    union
    {
        struct
        {
            vec3f Position = { 0, 0, 0 };
            quatf Rotation = { 0, 0, 0, 1 };
        } AsTransform;

        struct
        {
            byte State;
            byte Command;
        } AsAnimation;

        struct
        {
            vec3f   Position = { 0, 0, 0 };
            float   Duration = 0;
            int     TargetUUID = -1;
            SkillPacketTypes Type = SKILL_TYPE_UNKNOWN;
        } AsSkill;

        float AsFloatArray[7];

        float AsFloat;

        bool AsBool;
    }; // Package Specific Data
}
```

Figure 5.30 - Packet structure.

6 Asset Overview

Scare Tactics was originally going to be a 2D game. During our pre-production, we decided to change to 3D due to the nature and perspective of our game. Our game has two different camera perspectives. The *Ghost* has a broader view of the map while the *Explorers* have a much constrained view. In order to better show the depth, we made the decision to switch to 3D.

We wanted the game to have a simple and clean aesthetic. We were inspired by the simple geometry and the visual appeal of games like *Journey* and *Monument Valley*. This approach also helped us as the artists on our team weren't very familiar with organic modeling and we had multiple characters to create. The game is set in the 1900's. This was the time when technology was flourishing. The *Explorers* are a group of people that have come together from different parts of the world. Exorcising *Ghosts* isn't a job that everyone does and so they needed to create their own weapons and gadgets from the parts they could find.

In order to set a tone and direction for the aesthetic, we created a couple of mood boards, one for the characters (Figure 6.1), and another one for the environment (Figure 6.2).

Figure 6.1 - Character Mood Board.

Figure 6.2 - Environment Mood Board.

Over the next month, the concept artists designed characters (Figure 6.3, Figure 6.4 and Figure 6.5), environments (Figure 6.6) and the overall look and feel of the game while the 3D modelers started modeling smaller props and scene filler objects. Once the characters were modeled, our rigger/animator set up the rigs and created animations but due to time constraint and the restrictions posed by our C++ engine, we used *Adobe Mixamo* to create base rigs and animations for our explorers. From there, the animator took over and tweaked the rigs and animations to fit the aesthetic that we are going for.

Figure 6.3 - Professor Concept Art.

Figure 6.4 - Sprinter Concept Art.

Figure 6.5 - Trap Master Concept Art.

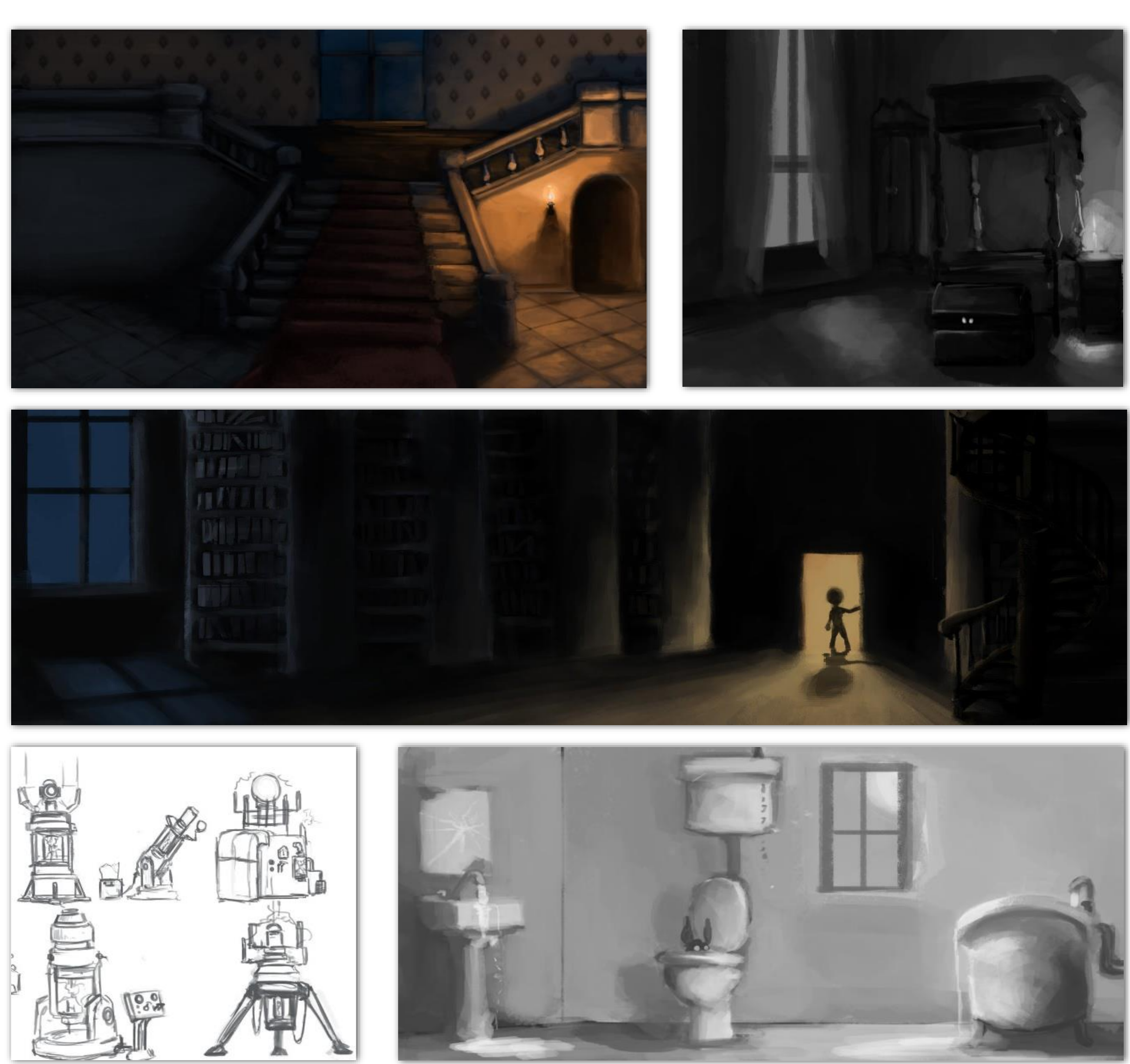

Figure 6.6 - Environment Concept Art. Entrance hall (top left). Master bedroom (top right). Library (middle). Generators and light-cannon (bottom left). Bathroom (bottom right).

Conveying the controls to the *Explorers* and the *Minions* to the *Ghost* needed more than words. This brought UI into the game. The icons and symbols were created to represent the characters and abilities as closely as possible (Figure 6.7). Some went through iterations after getting feedback from playtests. Check Appendix A for a complete asset list.

Figure 6.7 - UI icons.

7 Play Testing and Results

Our playtest sessions started out with faculty members and other classmates playing our game and verbally giving us their feedback. Once we had the basic mechanics nailed and our game got bigger, we held playtest sessions with people that had never played our game to get feedback on overall game design. We also held playtests with groups that had played our game before to get feedback on changes we made to existing mechanics and game balance.

Game Developer's Conference (GDC) 2016 was a big public playtest for us. Since our game requires 4 players for a single game, we decided to create an online questionnaire for players to fill out after each game. This allowed us to observe the players and explain the game to people waiting to play the game.

7.1 Internal

The internal playtests usually took place within the team or with other classmates weekly or biweekly. We asked the playtesters to focus on how the movement felt, if the controls were friendly and easy to use and if the core mechanic was fun to use.

We went through several iterations of keyboard controls and movement scheme based on the feedback we received. We altered the attacks for the *Explorers* to make it easier to hit the minions without losing health. The most important feedback we received from the internal playtest would be that the game felt like a race to get to objectives. The *Explorers* have an action - adventure style gameplay but we wanted them to strategize and win and not race against the *Ghost.*

7.2 Public

Our first public playtest was conducted with the artists working with us. At this point, we had our core mechanics nailed down. We were looking for feedback on the newer mechanics and a new level that was created. This level was much larger than the one we had been using until now. We wanted to

explore the aspect of exploration and player communication as the players playing as *Explorers* could easily get lost and lose without having each other's backs.

Surprisingly, the *Explorers* quickly figured out they needed to communicate to have a chance of winning and actually called out to each other, created strategies, warned others of dangers and asked for help. We realized that the new abilities were not easy to use without explanation and were either underpowered or way too overpowered. Additionally, a larger level meant longer times for *Explorers* to find objectives, which made it easier for the *Ghost* to win as he had the sight advantage and could plan his moves ahead of time.

7.3 Game Developer's Conference (GDC)

GDC was the biggest public playtest we participated in. Based on the feedback from our prior playtests, we had tweaked our mechanics to be as smooth as possible. One big change was adding indicators to other *Explorers* and all the objectives to give a general sense of direction to the *Explorers.* We also introduced three different floors in our level, giving the scene a deeper sense of depth, in spite of our game being 2D under the hood. These were the questions that were part of the online questionnaire:

1. Did you play as explorer, ghost, or both?
2. Describe your general strategy while playing the game.
3. How do the controls feel? Were they intuitive?
4. Which skills did you use the most?
5. What role did lanterns play in your decision making?
6. If you played as an explorer, did the gameplay encourage you to stay with other explorers? Why, or why not?
7. Would you play this game again? Why, or why not?

To our surprise, we received mostly positive reactions and people seemed to be having fun. Most of the core mechanics worked, but some needed our attention immediately. The *Explorers* did not use the lanterns and doors as much. It wasn't clear that they could be used as a mechanic. They did not know friendly fire existed. Eventually when they did realize, it was already too late. The hiding spots did not prove any use at all and the indicators weren't much help either. *Explorers* still felt lost and did not know where to go. On the other side, the *Ghost* was overpowered and almost impossible to defeat. The Imp minion was being used the most as it cost less, spawned two each time and were fast.

7.4 RPI GameFest 2016 / ImagineRIT 2016

RPI GameFest and *Imagine RIT* took place within two consecutive weekends. Since these events were so close to the end of the semester, we decided to not make any major changes to game and kept it the way it was.

RPI being more of a competition rather than a playtest event, we decided to only observe the players rather than take notes or making the players fill out a playtest feedback form. The UI changes and the addition of an instruction screen allowed us to examine the players without giving them instructions. They were able to understand the mechanics with very little supervision (Figure 7.1). This was a big step for us as we had been explaining the mechanics to every player. In the end, we also won 2nd place at RPI, which was a pleasant surprise to the entire team (Figure 7.2).

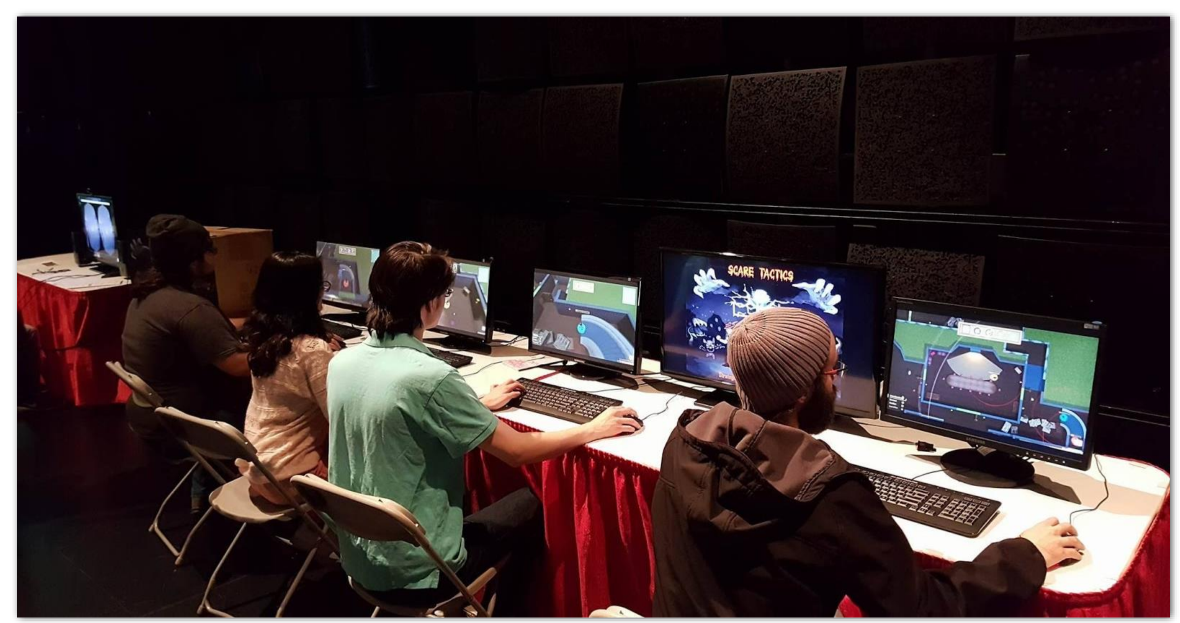

Figure 7.1 - RPI Playtest.

Figure 7.2 - RPI Award. From left to right:
Henrique Chaltein, Gabriel Ortega, Lucas Vasconcelos, Karan Sahu, Tiago Martines.

ImagineRIT was the first public playtest for our *C++* version of the game. We had playtested internally but never outside the team. We were all nervous if the *C++* build would survive for over 7 hours or would it crash too often, which would force us to switch to our *Unity* version. Surprisingly, the builds worked flawlessly with only 3 crashes over the entire day. Apart from a few minor bugs and slight balancing issues, people were able to play our game multiple times. The crowd at *ImagineRIT* was very different from what we had encountered in our earlier playtests. Most of the audience were either young children or non-gamers. The instruction screen wasn't enough most of the times and we had to explain the controls and the mechanics of the game.

7.5 Result

We compiled the feedback we received from *GDC* and went back to the drawing board to figure out solutions to the problems. We introduced a minimap in hopes of guiding the *Explorers* to the objectives. We also added an objective list as part of the UI to ensure *Explorers* know what they need to do from the get go. We removed friendly fire from all but one mechanic and switched the attacks for two of the *Explorers*. One of the biggest additions was a startup screen displaying instructions and controls for each *Explorer* and the *Ghost*. It also prevented the players from moving around in the game before everyone was ready.

When designing the game, our target audience was 13 and above due to the communication aspect between the *Explorers*, the different classes and the different minions that the *Ghost* can use. After *RPI* and *ImagineRIT*, we realized that plenty of younger children really enjoyed our game. They played the game multiple times and brought their friends to play it with them. This was a success in our eyes as it is difficult to please children. They would tell us straight to our face if the game was not fun, but they did not. They played for a long time, even the ones who were hesitant to play.

8 Post Mortem

8.1 Successes

In general, *Scare Tactics* has been received positively by those who have viewed and played the game. This can be attributed to several aspects of the *So Close* team's chemistry and production process. Aspects of note include the prototyping process, adoption of an *Agile* methodology, scheduling and commitment to improving the process throughout the project.

The initial eight weeks of prototyping served as a way for the team to learn how to work with one another. It allowed for individual team members to learn about one another's strengths and weaknesses. Working on smaller one week projects was good preparation for the scope and size of *Scare Tactics*. Throughout this period the team also gained experience working in *Agile* methodology and identified which parts of that methodology would be most useful during the development of *Scare Tactics*.

The constructs adopted by the team included the *Daily Standup, Weekly Sprint Planning, Demo, and Retrospective.* The *Daily Standup* provides an excellent way for team members to be aware of each other's work and stay informed of how his work influences the entire project. *Sprint Planning* allows for setting of weekly goals that can be broken down into smaller tasks. The *Demo* is a meeting where the team can assess the work done during the previous sprint, however, this meeting was discarded after a few weeks.

One of the most valuable aspects of *Agile* became the *Sprint Retrospective*, which is basically a weekly post mortem. During these meetings the team would identify sources of success, and areas of improvement. Once these areas were listed, a set of actionable policies would be set in place to improve the development process for the coming week. The retrospectives became a constant source of improvement for the team's communication, task delegation and goal setting.

All in all the team worked really well together and was able to build a project in *Scare Tactics* that they are proud of and has been well received.

8.2 Improvements

During the development process there were several areas where the team could have improved. Sometimes there were communication breakdowns between the art, design, and development teams. This proved costly in some cases because it lead to repeat work having to be performed, or misconceptions about the look, feel, and mechanics of a game. Some of these communication breakdowns could have been mitigated by a stronger focus on documentation. The team lacked a centralized source of information about the game being built.

The team also struggled with time approximations. Tasks routinely took longer than the time for which they had been scoped. In several cases this lead to falling behind schedule or features being dropped completely. This problem could have been mitigated by making better use of *Redmine* features. The *Redmine* Agile plugin provides mechanisms for tracking time spent on a task. Given more consistent use of this feature, the team could have used the data as a way to make better time estimates.

Lastly, the development pipeline could have been better. Ideally, a feature would have been implemented in *Unity* and then ported to the *Scare Tactics* engine. In practice, many features were implemented in *Unity*, while major systems were being built in the engine. This lead to the engine always having a larger weekly scope than the *Unity* prototype. At times it felt as if both versions were unrelated. This could have been improved by implementing some of the smaller game systems first in order to have a playtestable *C++* version at all times. For example, our animation system was implemented before the game state and logic that determines the winner and loser of the game. Had the game logic been implemented first the *C++* version could have been playtested much sooner.

8.3 Future Work

In regards to game design, we would like expand the world of *Scare Tactics* by creating 3 additional game modes, Escort, Escape and Hostage. We designed these as part of our original game design but were unable to implement them due to time restrictions. We would like to have new maps with differently themed environments such as an amusement park, scientific laboratory, and abandoned ship. This would add variety and hopefully increase the replayability of the game.

Our *Explorer* classes currently have 3 - 4 different skills, but only 1 - 2 unique skills. We would like to make the classes unique by adding abilities that allow for more varied gameplay. Since light acts as a mechanic in our game, we wanted to give each *Explorer* a different colored light that acts differently. Some other skills that were designed but could not be implemented are buff teammates, radar (reveals enemies and objectives as dots, similar to a ship radar), tesla coil (created by placing three rods in a triangle formation, damaging any enemy inside) and spring trap. It would give the players the option to create different strategies on the fly. Similarly, creating more minions for the *Ghost,* such as the ambusher, transporter and the poison, would give the player options to approach each level in a different manner as well.

We have visual feedback for the players, but they aren't as exciting as they could be. We would like to provide better visual and auditory feedback as it would make the gameplay more engaging. Originally, we planned to have some humor elements to it, which would be a nice addition, both in terms of audio and visual feedback.

Bibliography

Alexandrescu, Andrei. 2001. *Modern C Design: Generic Programming and Design Patterns Applied*. Boston, MA: Addison-Wesley.

Baggett, Dave. 2014. "Is C++ slower than C?" *Quora,* March 2014. https://www.quora.com/Is-C++-slower-than-C/answer/Dave-Baggett?srid=O175.

Champandard, Alex J. "Behavior Trees for Next-Gen Game AI." AiGameDev. December 28, 2008. Accessed April 18, 2016. http://aigamedev.com/insider/presentations/behavior-trees/.

Champandard, Alex J. "Understanding Behavior Trees." AiGameDev. September 6, 2007. Accessed April 18, 2016. http://aigamedev.com/open/article/bt-overview/.

Cossell, Stephen, and Jose Guivant. "Parallel evaluation of a spatial traversability cost function on GPU for efficient path planning." *Journal of Intelligent Learning Systems and Applications* 3, no. 04 (2011): 191.

Durant, Sidney. "Understanding Goal-Based Vector Field Pathfinding." Game Development Envato Tuts. July 5, 2013. Accessed April 18, 2016. http://gamedevelopment.tutsplus.com/tutorials/understanding-goal-based-vector-field-pathfinding--gamedev-9007.

Ericson, Christer. 2005. *Real-time Collision Detection*. Amsterdam: Elsevier.

Gamma, Erich. 2011. *Design Patterns: Elements of Reusable Object-oriented Software*. Upper Saddle River, NJ: Pearson Education.

Green, Chris. "*Improved Alpha-tested Magnification for Vector Textures and Special Effects.*" ACM SIGGRAPH 2007 Courses on - SIGGRAPH '07, 2007.

Gregory, Jason. 2009. *Game Engine Architecture*. Wellesley, MA: K Peters.

Harabor, Daniel Damir, and Alban Grastien. 2011. "Online Graph Pruning for Pathfinding On Grid Maps." In *AAAI*.

Luna, Frank D. 2012. *Introduction to 3D Game Programming with DirectX 11*. Dulles, VA: Mercury Learning & Information.

Millington, Ian, and John David Funge. 2009. *Artificial Intelligence for Games*. Burlington, MA: Morgan Kaufmann/Elsevier.

Nystrom, Robert. 2014 *Game Programming Patterns*.

Appendix A - Asset List

1. Explorers
    a. Sprinter
    b. Professor
    c. Trap Master
2. Ghost Minions
    a. Imp
    b. Abomination
    c. Flytrap
3. Environment
    a. Wall
        i. Square
        ii. T-intersection
        iii. L-corner
        iv. Wall with single door space
        v. Wall with double door space
        vi. Wall with window space
    b. Floor
    c. Lantern
        i. Portable
        ii. Wall
    d. Stair
        i. Straight
        ii. Curved

e. Couch

    i. Single seater

    ii. Double seater

f. Chair

    i. Dining

    ii. Study

    iii. Office

g. Table

    i. Dining

    ii. Study

    iii. Round living room

    iv. Rectangular living room

    v. Office

h. Bed

    i. House owner

    ii. Staff

i. Cabinet

    i. Bathroom

    ii. Dining room

j. Bookshelf

    i. Straight

    ii. Curved

    iii. Flat

k. Door

    i. Single

ii. Single blocked

l. Stack of books

m. Trap box for glue and poison

n. Light cannon

o. Generator

p. Bedside Drawer

q. Sink

r. Toilet

s. Bathtub

t. Fireplace

u. Mirror

v. Lever

w. Bottles

x. Wood planks

y. Boxes

z. Wardrobe

4. UI

a. Explorer Icons

i. Baton Bash

ii. Staff Swing

iii. Grenade

iv. Heal

v. Sprint

vi. Poison Trap

vii. Glue Trap

viii. Portable Lantern

ix. Sprinter Indicator

x. Professor Indicator

xi. Trap Master Indicator

xii. Attack Mouse Cursor

xiii. Interact Mouse Cursor

b. Ghost Icons

i. Imp

ii. Abomination

iii. Flytrap

iv. Imp Illusion

v. Mana Bar

vi. General Mouse Cursor

vii. No Spawn Mouse Cursor

c. Text background

d. Splash Screen