

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Table of Contents

PITCH DOCUMENT	3
ELEVATOR PITCH	4
HIGH CONCEPT DOCUMENT	5
GAME TREATMENT DOCUMENT	7
PITCH PRESENTATION.....	13
GAME DESIGN DOCUMENT	30
LEVEL DESIGN DOCUMENT	91
LEVEL I: FORT ASHFORD	92
TECHNICAL DESIGN DOCUMENT	96
ART BIBLE	162
INDIVIDUAL RESEARCH DOCUMENTS	176
MATTHEW BOZARTH: DEFERRED SHADING	177
COLIN DOODY: A GENERIC USER INTERFACE SYSTEM FOR GAMES	181
ANDREW GALANTE: BEHAVIOR TREES IN GAME AI	191
JOSHUA GILPATRICK: NEURAL NETWORKS.....	195
NICHOLAS KORN: MULTI-THREADING GAME AI	198
GAMEPLAY PROTOTYPE REPORT	202
TECHNICAL PROTOTYPE REPORTS.....	206
MESH FILE FORMATS TECHNOLOGY TEST	207
ARTIFICIAL INTELLIGENCE CAPABILITIES TEST	212
USER INTERFACE TECHNOLOGY TEST	216
APPENDIX	220
APPENDIX A: ASSETS LIST	221
APPENDIX B: THIRD PARTY REQUIREMENTS.....	234
APPENDIX C: SCHEDULING INFORMATION	238

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Pitch Document

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Elevator Pitch

Imagine rolling onto the battlefield in your favorite tank. Friendly forces gather up and join you, ready for the attack. The enemy has been pushing forward into your territory but you are prepared to fight back. The battlefield a frozen tundra, of trees, rocks, boulders and city remains lies before you. As your friendly forces proceed to attack from the South you roll off the East and begin to weaken the enemy from the side. Using the surrounding environment to your advantage, you are able to avoid enemy fire and navigate around their forces. While the battle progresses, it is your duty to both assist your friendly forces as well as complete objectives. You will be asked to assault enemy positions, destroy major targets and even protect friendly assets.

Unlike many games, where the player's progress is set linearly, this game lets the players choose what order they wish to complete objectives. Not only will the player have the ability to approach the battlefield in their own way, they will also have the option of choosing a tank best suiting their needs. Up to three of your friends can join your forces to defeat the enemy. Work with your team mates and approach the battle in new ways.

High Concept Document

Cooperative, tactical tank combat in a living battle.

Short Description: Your army needs you as they push deep into enemy territory. Fight alongside your friends and the computer as you play through a dynamic campaign. The enemy can attack you from anywhere, so think quickly and outsmart them.

Tone words: action, fast paced, tactical, teamwork

1 Game High Concept

Select from four tanks the one which best fits your play style. This will influence the way in which you approach the upcoming battles. Whether you enjoy being in the middle of the action or staying back and picking off the enemy from afar, there is a tank to match your needs. Your friends can join in too, choosing what best fits them.

On entering a mission, you and your friends are given a list of objectives to be completed. When and how you complete these is completely up to you. Working with team mates and allied NPCs, you can begin to move toward your first objective. Cooperate with your friends and flank enemy units using the surrounding environment for cover. If you prefer to be more covert, position yourself to ambush the enemy unit and catch them by surprise.

What separates us from the competition is the ability to play with not only your friends, but to work with a large army of friendly NPC units.

2 Development Scope

The shipped game will include two maps for users to play on. Each of these maps will consist of several objectives and take anywhere from 10-30 minutes to complete. Players will have a selection of four different tanks to choose from in their networked cooperative play. The maps will be balanced for single player and multiplayer play. Each map will have a range of difficulty the player can choose from: easy, medium, and hard.

3 Business Case

Inspired by games like Iron Tank, Gauntlet, and other classic arcade campaign games that are seeing a return through digital markets, this game offers a similar cooperative experience, but with a fresh depth of teamwork and tactics available to the players as they choose their path. Our game is in good company with the recent releases of games such as Castle Crashers (a 1-4 player coop fantasy game) with over

350k copies sold [1] [2] and Assault Heroes 1 with over 250k [3] copies and Assault Heroes 2 also performing well.

The team developing the game is made up of five master's students in the Game Design & Development Masters program at the Rochester institute of Technology.

Each member of the team has a specialized skill that makes the development of this game possible. Joshua Gilpatrick has a networking background, and is able to build the networking protocol to sync multiple game clients together that allows friends to dominate the enemy together. Colin Doody is currently involved with his own start-up and will offer team management and leadership, as well as a keen eye for an artistic design. Andrew Galante has worked both sides of the fence learning about the graphics side of game programming and the artificial side, and will be assisting in the development of the artificial intelligence required for this game. Matthew Bozarth takes a keen interest in design and has spent his time in academia researching engine structure and performance. The final member of the team, Nicholas Korn, has a passion for multi-threading and will be using these skills in development to allow for a more complex game environment that takes advantage of multi-core systems.

4 References

[1] VGCharts Staff. "XBLA Arcade Sales Charts". Internet:
<http://news.vgchartz.com/news.php?id=2569>, Nov. 22, 2008.

[2] Kim Fidler. "Castle Crashers Sells 250k, Alien Hominid Sales Increase". Internet:
<http://360.kombo.com/article.php?artid=14319>, Sept. 16, 2008.

[3] VGCharts Staff. "Xbox Live Arcade Sales Charts". Internet:
<http://news.vgchartz.com/news.php?id=929>, Mar. 22, 2008.

Game Treatment Document

Command your tank, work with your teammates, and out-maneuver the enemy.

A Game Treatment

1 Overview

The inside of your tank is hot and dry, just like it is outside only more so. You sit restlessly with nothing to do but wait. Then the call comes, B Company is in position and you are to meet them at the bridge and proceed to the factory destroying any opposition you find. Firing up the motors a warm and deafening purr envelops your world as you slide on your helmet and begin plotting a course. Your tank is sort of a unique vehicle. It has what most every tank has: two sets of treads, armor, some big guns, and some enormous motors to power its immense weight. But the weapons are something special. Electrical powered rail guns whose rate of fire is restricted only by the amount of power available. Yours is set up to provide a quick burst of several shots before recharging. The idea being that you can kill anything before you temporarily run out of ammo. If not, you still have those powerful engines that are currently taking you closer to battle.

Tone Words: action, fast paced, tactical, teamwork

2 Game Concept

There is a battle going on and you are involved as a single tank driver. It is your duty to assist the main battle force in their objectives whether that is assaulting an enemy position, destroying a target, or protecting friendly assets. For your convenience, several different types of tanks have been made available to you. Choose between fast and lightly armored or slow but well-armored. Do you want a heavy hitting gun with a slow rate of fire or something better suited to picking off groups of enemies? These decisions are yours to make. Keep in mind that you aren't fighting alone. A slow moving tank with little armor and a really big gun might sound like a bad idea, but if you can take advantage of friendly troops to let you rain fire from afar this could be an effective tank. Mix and match tank classes when you play with friends as well. Levels are designed to allow 1-4 players with a variety of tanks on the battlefield.

Enemy and non-player friendly tanks will be controlled by an artificial intelligence focused on providing the player with tactical options while also focusing on projecting a variety of personalities. Your tactics must change depending on what kinds of enemies you face; not only due to their firepower.

Players will progress through a single campaign made up of several levels. In each level, players will have several objectives to complete. Unlike many games where players progress linearly along a path, this

game will let the players choose in what order they want to complete objectives. Because of this, players will also be able to approach a single objective from several different directions. This along with the friendly forces who will be fighting alongside, each level will offer much replay value.

The world is not a sandbox however; the enemy tanks as well as friendly forces are scripted to interact intelligently with the player. A player could approach a bridge with a small force of friendly troops to find the enemy bunkered in. As they approach the bridge a small group of enemies may sweep in behind them cutting off any escape route. This situation looks hopeless for the player until friendly troops arrive on the other side of the bridge to rescue them. While a situation like this could possibly occur in a true open-ended environment, this game will utilize a scripting system to help guide these battles.

3 Critical Path

The player will be required to complete or assist in completing objectives necessary to win the battle. Since the game world is open-ended, the specific route and tactics employed by the player aren't constrained. There will be a small number of possible conditions for completing an objective – such as moving to a prescribed point, killing a specific unit, or removing enemy defenders – but the choice of how to achieve these conditions is left up to the player.

A possible scenario: as you move across the battle field, you take advantage of the rocks, trees, and other units around you. Use them as protection from the enemy as you position yourself at key points in the field to open fire and pick apart the enemy's front line. Open a hole in their defenses to allow your troops through on to complete the objective.

4 Features and Controls

This game will be designed to be played with the keyboard and mouse.

Players are controlling tanks and these will behave realistically. Tanks will be able to turn in place due to their dual-treads and will have a noticeable momentum to them.

These controls are only a default setting and will be modifiable by the player.

Move Forward: W

Turn Left: A

Turn Right: D

Back Up: S

The tanks turret will be controlled using the mouse as a reticule. The turret will have a maximum turning rate so the player will need to take the turrets current rotation into account before firing.

Each tank will have two weapons: a light machine gun useful for destroying small enemies or slowly whittling away at heavily armored foes, and a main gun used to destroy heavily armored targets. The

light machine gun will fire rapidly but will be prone to overheating and players will have to let it cool off after heavy use. This weapon has the benefit that it is always aiming in the direction of the reticule.

The heavy gun will function differently depending on the tank. This gun is powered by an energy pool that every tank has. Firing the heavy gun will drain the energy tank. If there is not enough energy, the gun will not fire. Tanks will regenerate energy over time. The amount of energy used in firing and the rate of energy regeneration varies from tank to tank. All tanks will have a shield that is powered by a separate generator. This will hold up under light fire but will fall under heavy damage. It will reactivate after a tank is outside of combat for a short period of time.

Here are some examples of the different tanks and how they manage their energy.

Rapid Fire Tank

Speed	Fast
Armor	Low
Damage	Low
Cool-down	Medium
Energy Usage	75%
Energy Regeneration	Very high

The high regeneration rate of the tank allows it to fire very fast and regenerate quickly. It quickly runs out of energy, but it's regeneration rate is fast enough that it can recover rapidly and return to action.

Battle (Standard) Tank

Speed	Medium
Armor	Medium
Damage	Medium
Cool-down	Medium
Energy Usage	25%
Energy Regeneration	Medium

This tank is able to fire four times before it must wait for its energy to refill.

Heavy Tank

Speed	Slow
-------	------

Armor	High
-------	------

Damage	High
--------	------

Cool-down	High
-----------	------

Energy usage	50%
--------------	-----

Energy Regeneration	Low
---------------------	-----

This tank is able to fire twice before needing to wait for energy to recharge. It is also able to regenerate energy faster if it is not moving.

Burst Tank

Speed	Medium
-------	--------

Armor	Medium
-------	--------

Damage	Medium
--------	--------

Cool-down	Low
-----------	-----

Energy Usage	10%
--------------	-----

Energy Regeneration	Low
---------------------	-----

This tank can fire 10 rounds very quickly, but it takes a lot of time for it to recover afterwards. This tank will regenerate energy if it's being attacked.

5 Interface

A large part of the game is playing with friends and fighting alongside friendly troops. The interface will reflect this by giving you information about your friends and showing you where friendly troops are in the vicinity.

6 Development Scope

This game will feature two missions. Each mission will have several objectives and will take anywhere from 5-20 minutes to complete.

7 Technology Features

We will be writing our own 3D engine for this game.

- Artificial Intelligence
- User Interface
- Deferred Lighting
- Networked Multiplayer
- Immersive Audio Experience

8 Potential Technology Roadblocks

The believability and fun interaction with the AI will prove critical to this game's success. Research into how the AI will be implemented has already begun.

9 Platform

The game will be available for Vista, developed using DirectX 10 and C++.

10 Business Case

Our team consists of five individuals who have unique skills and backgrounds. Each individual has a bachelor's degree in a computing field, and is pursuing their masters in Game Design and Development. Two members of the team have experience working in the game industry developing for the Xbox Live platform, while another member started their own software company. The specialties of the team range from 2D/3D animation, networking, artificial intelligence, to engine design.

This game will target audiences twelve and older; the reason for this is the semi-realistic violence may not be appropriate for children younger than this age. As for adults, the gameplay keeps them engaged with strategy and tactical planning to accomplish the game. Games that fit into this genre fit into ~37% of the PC sales market.

Inspired by games like Iron Tank, Gauntlet, and other classic arcade campaign games that are seeing a return through digital markets, this game offers the same cooperative experience as those, but with something new with regards to the depth of teamwork and tactics available to the player as they choose their path. Our game is in good company with the recent releases of games such as Castle Crashers (a 1-4 player coop fantasy game) with over 350k copies sold and Assault Heroes 1 with over 250k copies and Assault Heroes 2 doing well also.

<http://news.vgchartz.com/news.php?id=929>

<http://news.vgchartz.com/news.php?id=2569>

<http://360.kombo.com/article.php?artid=14319>

Players in our target audience often play games that require some kind of teamwork. Either with other players or with NPC's, example of a games written that also take advantage of this are Left 4 Dead, and the Battlefield series. Our audience will also have an interest in games that have a tactical or strategic element to them. Games in this category are often real time strategy games.

11 Competitive Analysis

11.1 Assault Heroes 1 & 2

The Assault Hero games are top-down shooters for the Xbox Live Arcade system. One to two players traverse a linear map in a tank like unit blowing up everything that gets in their way. Tanks are equipped with an assortment of deadly weapons consisting of a mini-gun, missile launcher, flame thrower, freeze gun, nuclear explosion and grenades. Throughout the course of the game other vehicle types are made available to the player. These are relinquished at the end of an area and the player goes back to the original tank unit.

Most of the weapons in the game have limitless ammunition. This allows for the enemies to be very abundant and create a hectic environment. For Assault Heroes this style works very well. Our game will differ by having a limit on how frequently the player can fire. This forces the players to think more tactically before taking each shot. The hope for our game is not to create a hectic environment but one where the player must be more tactical.

Players can play cooperatively on the same Xbox or through Xbox Live. While playing in co-op mode the players are forced to stay within the same screen space. This forces players, who may usually forge ahead, to stick with their team mate. Because the levels are very linear this co-op model works very well. Our game will improve on this by allowing players to split up and cover a larger area of the level. This will work better for our game as it is a more open level design meant for the players to take advantage of the maps layout.

The variety of enemy types in Assault Hero is sorely lacking. Although visually the game has many unit types there are very few different functional types of units. The game can be broken down in to small gunners, kamikaze, tanks and bosses. These units all use a very simple technique of running towards the player and then once in range begin shooting. The bosses are the only unit type which differs from this technique. Bosses act based on a set pattern which repeats throughout the battle. Each unit in our game will have a specific tactic associated with its type. The player must use different tactics when approaching each battle situation. This removes the receptiveness between each of the battles.

11.2 Wii Tanks

Wii Tanks is a top-down tank game for the Wii console. The game can be played as either single or two player co-op. Players are pitted against enemy tanks in an enclosed single screen area consisting of different layouts. In order to proceed to the next level the players must defeat all of the enemies in the current level.

Artificial intelligence within Wii Tanks is very simple. The units are either sitting still and aiming at the player or wandering around the level avoiding the player while trying to make a shot at the player. The entertainment value in the game comes from the level design more than how the enemy acts. Our game will improve on the Wii Tanks method by creating more diverse enemy behaviors. This will create different situations the player will have to confront in order to succeed.

Wii Tanks is limited to a single screen of play space. The layout of the level and placement of enemies is very critical to how much the player will enjoy a level. Wii Tanks is very successful with its level design and provides a fun experience for the player. Our game hopes to expand on the well laid out levels of Wii Tanks to a larger level design. The player will experience a more expansive game world drawing him or her further into the game.

Players also have a limited amount of fire power at their disposal in Wii Tanks. Each player can have up to 5 bullets on screen at any given time. This forces the player to think critically about how to place the shot in order to minimize periods of time when shots are unable to be used. Our game will follow this technique by limiting how frequently the player is able to shoot their weapon based on their tanks class.

11.3 Iron Tank

Iron Tank is a single player top-down tank game for the NES. The player is faced with a linear single player campaign against a computer army. The player is able to fight off enemy forces with two different weapons, the main cannon and a small machine gun mounted atop the tank. The machine gun is rarely useful as most everything requires the main cannon in order to be destroyed. The player has no ability to move backwards in the game world only allowing for forward and some side to side movement.

Iron Tank does an excellent job of portraying the enemies' army as the player moves throughout the campaign. The player gets the feeling they are single handedly battling a full scale army without feeling as though they are completely out numbered. Our game expands on this idea of the enemy having a full scale army by having the player support his/her own army. The player will support their team and change the outcome of the battle in their favor. We will create the feeling for the player such that if they are the turning point of a losing battle.

Pitch Presentation

Matthew Bozarth, Colin Doody, Andrew Galante, Joshua
Gilpatrick, Nicholas Korn

TANK GAME
TITLE TBD



Your objective is to take out the two radio towers. One objective is to the North located in the old downtown area. The other is located to the West in a field.



You will approach from the South. Friendly forces will work to meet up with you from the West and the North West.



Enemy forces meet you at the river crossing and start an assault.



Additional enemy forces arrive from the East.



Friendly forces come from the North in order to help eliminate the enemy threat.



As you proceed to the North, friendly forces proceed East.



Splitting off from the group headed east a second team heads west in order to take out the radio tower.



Regrouping with the group from the west you encounter more enemy units.



Working with the friendly tanks you take out the enemy forces and the final radio tower.

Game Overview



Our game is about controlling a single tank in a war time environment. You will be handed objectives from your superior and complete them in a real time game world. You are responsible for steering and moving your tank across the battlefield, firing at enemies, staying alive, and completing the mission objective. This will all take place in a relatively open environment, so the tactical decisions to be made are yours. Allied units will be battling around you, but your skills will be required to tilt the scale in this war.

Features

- 1 – 4 Players (Cooperative)
 - Over a network
- Objective-based Campaigns
- Non-linear Level Design
- Autonomous Friendly Units
- Multiple Player / Tank Classes

1-4 players can play cooperatively over a network. Campaigns will consist of a series of objectives which will take place within an environment. The levels are design to be non-linear. We want to have a “sandboxy” environment in which the player can make original tactical decisions. Friendly units will act autonomously on their own accord. While their introduction may be triggered by a certain event, their actions will be driven by their AI. We will also be supporting multiple tank classes for the user to choose from. Each tank will have benefit over the other, allowing users to choose their preferred play style.

The Experience

- Tactical Freedom
- Influence the battle
- Active environment

The player will have freedom to decide how they want to approach a battle. The player's actions within the world will have an influence on how the battle unfolds, and how and if the objectives are completed. Even though the player plays an active and critical role in how the battle evolves, it will also be apparent that the world is operating with or without the user's participation.

Development Scope

- One campaign
 - Two Levels
- Four user classes
- Level editor

There will be one campaign consisting of two missions. The game will be developed with four different classes to accommodate the four players. Due to the dependence on the world in our gameplay, and the “scripted” sequences that may occur, we are going to need a “level editing environment”, whether it’s a separate application or not. This will allow us to tweak gameplay easily and create unique experiences throughout a level.

Development Challenges

- Level Design
 - Content Creation
- Balancing
- Artificial Intelligence
- World Simulation
- Scripted Game Events

Level design

- Designing a level will be a time consuming experience
- Will be challenging to make an enjoyable experience
- Creation of content will be difficult and time consuming

Balancing

- Game will need to be balanced from an AI perspective and from a level design perspective

Artificial Intelligence

- Gameplay relies heavily on the interaction of the player and the friendly and enemy AI units.
- Ensuring that the AI is appropriately challenging will be difficult

World Simulation

- Technically challenging. Need to simulate more than just what the player sees.

Scripting

- Need to develop a game engine that supports these in game experiences. Gameplay doesn't arise just from the interaction of the AI and the player. Events need to occur in the game and our engine needs to be dynamic enough to support these sequences.

Capstone

- Demonstrates aptitude for entire game development process
- Suitable graphics
- Culmination of our studies

We believe this is a good capstone project as it includes various aspects of game design and development. It requires good level and gameplay design tuning and creation. It also requires a good understanding of 3D development as well as optimization techniques. It will require well thought out artificial intelligence.

The geometry is fairly simple for the game and does not require high resolution graphics. It will allow for the use of many particle systems.

This game combines not just technical skills that we have learned in the program over the years, but also integrates a practice in game design which we have not done in the past.

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Game Design Document

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Table of Contents

1	GAME OVERVIEW	34
2	WORLD.....	36
2.1	MISSIONS	36
2.1.1	<i>Stages of a Mission</i>	<i>36</i>
2.1.2	<i>Objectives.....</i>	<i>36</i>
2.1.3	<i>Difficulty Levels</i>	<i>37</i>
2.2	ENVIRONMENT	37
2.2.1	<i>Terrain.....</i>	<i>38</i>
2.2.2	<i>Environmental Objects.....</i>	<i>38</i>
2.3	EVENTS	38
2.3.1	<i>What an Event Is.....</i>	<i>39</i>
2.3.2	<i>What Causes an Event</i>	<i>39</i>
2.3.3	<i>Where Events Happen.....</i>	<i>39</i>
3	TANKS	40
3.1	MOVEMENT AND CONTROL.....	40
3.2	SHIELDS.....	41
3.3	WEAPONS	41
3.3.1	<i>Projectiles.....</i>	<i>41</i>
3.3.2	<i>Artillery</i>	<i>42</i>
3.4	FRIENDLY FIRE	42
4	PLAYER	43
4.1	LIVES AND DEATH.....	43
4.1.1	<i>Lives</i>	<i>43</i>
4.1.2	<i>Death and Rebirth.....</i>	<i>43</i>
4.1.3	<i>Death Without Rebirth.....</i>	<i>44</i>
4.2	ATTACK AND DAMAGE	44
4.2.1	<i>Primary Weapon.....</i>	<i>45</i>
4.2.2	<i>Secondary Weapon</i>	<i>45</i>
4.2.3	<i>Minimum Attack Range</i>	<i>46</i>
4.2.4	<i>Energy.....</i>	<i>46</i>
4.3	PLAYER TANK CLASSES	46
4.3.1	<i>“The Hornet”</i>	<i>48</i>
4.3.2	<i>“The Rhino”</i>	<i>49</i>
4.3.3	<i>“The Lancer”</i>	<i>50</i>
4.3.4	<i>“The Ant”</i>	<i>51</i>
5	NON-PLAYER CHARACTERS	52
5.1	SPAWNING	52

5.2	FIRING	53
5.3	FRIENDLY UNITS.....	53
5.4	ENEMY UNITS.....	53
5.5	NPC BEHAVIOR	54
5.6	NPC UNITS.....	55
5.6.1	<i>Light Tank (Wolfhound)</i>	55
5.6.2	<i>Battle Tank (Durant)</i>	57
5.6.3	<i>Artillery (BRAMM)</i>	58
5.6.4	<i>Heavy Tank (Stonewall)</i>	60
5.6.5	<i>Turrets</i>	62
6	GAME STATES / FLOW.....	63
6.1	LOGO SCREEN.....	63
6.2	MENU SCREEN	64
6.2.1	<i>Start Solo Mission</i>	64
6.2.2	<i>Start Team Mission</i>	64
6.2.3	<i>Game Options</i>	64
6.2.4	<i>Credits</i>	64
6.2.5	<i>Exit</i>	64
6.3	GAME OPTIONS	65
6.3.1	<i>Graphics</i>	65
6.3.2	<i>Controls</i>	65
6.4	SOLO GAME SETUP	66
6.4.1	<i>Player Class</i>	66
6.4.2	<i>Mission</i>	67
6.4.3	<i>Difficulty</i>	67
6.5	TEAM GAME SELECTION	68
6.5.1	<i>Game Names</i>	68
6.5.2	<i>Creating a Game</i>	68
6.5.3	<i>Joining a Game</i>	69
6.6	TEAM GAME SETUP	70
6.6.1	<i>Configuring the Game</i>	70
6.6.2	<i>Starting the Game</i>	70
6.6.3	<i>Chat Box</i>	70
6.7	MISSION INFO / GAME LOADING	71
6.7.1	<i>Team Game Considerations</i>	71
6.8	GAME	72
6.8.1	<i>Status Panel</i>	72
6.8.2	<i>Map</i>	74
6.8.3	<i>The In-Game Menu</i>	74
6.9	MISSION STATISTICS	75
6.9.1	<i>Overall Mission Statistics</i>	75
6.10	CREDITS	77
7	CAMERA (PLAYER VIEW)	78

7.1	MISSION INTRODUCTIONS.....	79
7.2	DEATH SEQUENCE	79
8	AUDIO	80
8.1	SOUND EFFECTS.....	80
8.1.1	<i>User Interface Sound Effects.....</i>	<i>80</i>
8.1.2	<i>Action / World Sound Effects</i>	<i>81</i>
8.1.3	<i>Environmental Sound Effects</i>	<i>82</i>
8.1.4	<i>Notification Sound Effects.....</i>	<i>82</i>
8.2	DIALOGUE	83
8.2.1	<i>General Dialogue</i>	<i>83</i>
8.2.2	<i>Mission Specific Dialogue.....</i>	<i>84</i>
8.3	MUSIC	84
9	MULTIPLAYER EXPERIENCE.....	86
9.1	ALIAS.....	86
9.1.1	<i>Conflicts in Alias Names.....</i>	<i>86</i>
9.1.2	<i>Predefined Alias Names</i>	<i>86</i>
9.2	COMMUNICATION	87
9.3	LEAVING THE GAME.....	87
9.3.1	<i>Team Leader Disconnection</i>	<i>87</i>
9.3.2	<i>Teammate Disconnection</i>	<i>88</i>
10	CONCLUSION	89
11	REFERENCES	90

1 Game Overview

Imagine rolling onto the battlefield in your favorite tank. Friendly forces gather up and join you, ready for the attack. The enemy has been pushing forward into your territory but you are prepared to fight back. The battlefield a frozen tundra, of trees, rocks, boulders and city remains lies before you. As your friendly forces proceed to attack from the South you roll off the East and begin to weaken the enemy from the side. Using the surrounding environment to your advantage, you are able to avoid enemy fire and navigate around their forces. While the battle progresses, it is your duty to both assist your friendly forces as well as complete objectives. You will be asked to assault enemy positions, destroy major targets and even protect friendly assets.

The player controls a tank and completes missions in large, hostile environments. Each mission can consist of several offensive or defensive objectives that the player must complete in order to complete their mission. The enemy is tasked with preventing the player from completing his goals while friendly units help the player, creating a large battlefield filled with action. Friends can join in the action, opening new tactics and options when approaching enemy units. Utilize your friends to flank from the sides or give cover fire as you dive in to the middle of an enemy group.

Play focuses around the ability of the player to attack enemies and dodge incoming projectiles. Slow moving projectiles give the player the ability to dodge, use surrounding cover, and also force the player to lead their shots giving depth and requiring skill to aim well. This also opens up more interesting tactics as players will benefit from engaging in strafing runs as opposed to driving straight towards the enemy where they won't have time to turn and dodge incoming fire.






Setting






The game takes place in a future when frozen white tundra covers the surface of earth. What little civilization is left has broken into factions and now fights for food and farmable land. Clear skies give way to the last remaining sunlight and full brightness of the moon. Cannon fire and left over street lamps give bursts of light to the battlefield.

Devastation from the battles has left dark scars in the tundra exposing rocks and boulders. Ruined concrete cityscapes now provide excellent cover as well as impassible terrain for armies to take tactical and strategic advantage of. Old concrete skyscrapers now lie in ruin showing only the base of their once grand structure. Pine tree coverage is dense in some areas creating tight pathways and choke points. Rocky cliffs and chasms surrounding areas create additional natural boundaries. Rundown factories and power plant monoliths are relics of an older, more civilized age.






Sample Color Palette:







Dull	
	#334455
	#447799
	#6699bb
	#99bbcc
	#887766

Vibrant	
	#334466
	#0066cc
	#0088ff
	#88ccff
	#aa7766



Dull	
	#220000
	#551111
	#aa5511
	#dd9922
	#ffeeee

Vibrant	
	#220000
	#770000
	#dd0000
	#ff7700
	#ffffff

created using the DeGraeve Color Palette Generator [1]

2 World

The world is a combination of a single mission, events in that mission and the environment in which it takes place. The environment is a description of the terrain the mission takes place on as well as the objects both fixed and destructible placed on the terrain. A mission takes place within an environment and has a set of objectives which need to be completed before the mission is finished. As part of the mission the player will experience mostly free-form battle but occasionally an event will occur which has been pre-defined.

2.1 Missions

When playing the game, a player is responsible for carrying out a mission. A mission consists of any number of objectives within a single environment. In order to fully complete a mission, these objectives must be completed. The player may choose to complete the objectives in any order that they see fit. A mission is failed when either a player runs out of lives or when an objective is failed.

2.1.1 Stages of a Mission

Start of a mission: A mission starts with the introduction of objectives accompanied by a camera sequence illustrating key areas of the map. A player has no control over the camera or their unit at this time. The camera moves across key areas of the map and text in the middle of the screen fades in and out to describe these key elements. A faceless commander also speaks over the radio, explaining each objective as it fades in and out. The fading in and out of a key area occurs over a short period of time and each key location is highlighted for enough time to understand the locations layout. This sequence can be skipped, leading a user directly into the action. In a multiplayer situation, only the host or team leader is able to skip the sequence.

During a mission: After the introduction of a mission, the player gains control of his vehicle and will maintain this control until the mission is completed, the game is exited, or the player dies and decides not to continue.

After a mission: A mission ends immediately after the last objective is accomplished, regardless of the player's situation in the game. The game halts at this point and the player is presented with text informing them that they have completed the mission. This is displayed long enough for the players to realize the win condition has been met. The players have the option to skip this sequence. Post game statistics are presented to each player after the mission accomplished text has been displayed and faded away.

2.1.2 Objectives

An objective within the game is a single accomplishment or a series of accomplishments that players or friendly NPC units are able to complete. The point of any given mission is to stay alive and complete these objectives. When all objectives are completed, the mission is won. If an objective is failed, the mission is lost.

Some objectives can be completed by friendly NPC units, while others are only accomplished by a player. The destruction of ten buildings around a map, for example, might be an objective that friendly NPC units will attempt to complete and can complete without the player's assistance. On the other hand, an objective like remaining within a region for a certain amount of time is one that can only be initiated by a player and can only be accomplished by a player within the mission. Which objectives are player-only and which ones can be completed by either a player or friendly forces depends on the mission, but in general the unique objectives will be player only, and broader objectives that can be incrementally completed over the course of the battle, such as destroy X towers, can be completed by either.

2.1.2.1 Examples of objectives

The following list is a summary of the objectives that can exist within the game. Each mission specifies the objectives that it implements and a mission does not need to include all types of objectives.

- **Destroy a target structure:** A destructible object or several destructible objects exist within the world that needs to be destroyed. Once all objects are destroyed, the objective is complete.
- **Remain alive within an area for a limited amount of time:** This requires a player to remain alive within a region for a certain period of time. The region and the amount of time that a player must remain within that area are specific to the mission itself. One player must exist within the region in order for the objective to be completed. If there are no players within the region, then the objective is failed. This does not include friendly units.
- **Defend a structure for a limited amount of time:** A user must protect a structure for a limited amount of time. If the structure is destroyed, then the mission is failed.
- **Destroy a target enemy unit:** A user may be required to destroy a target enemy unit. In this situation, a target enemy unit will exist uniquely within the world. The behaviors of that unit are described by the mission itself. A mission cannot be completed until the unit is destroyed. The unit may be mobile, so the objective itself will also be mobile on the map and visible to the user.

2.1.3 Difficulty Levels

All missions have multiple difficulty levels that they can be played with in order to create a compelling experience for players with various skill levels and to increase replayability. The difficulty level of a mission is set at the beginning of a mission. The game supports three difficulty levels: **Easy**, **Normal**, and **Hard**. Each difficulty level modifies the amount of damage that the enemy units do to the player and friendly units and the number of enemy units that are in the world in each mission.

2.2 Environment

The environment is a combination of the terrain and environmental objects where a mission occurs. The environment is generally large enough that it would take a player a few minutes to move from one end of the map to the other. Although environments are organic in shape, they are generally no wider than they are tall, in order to remove the feeling of a linear experience. The environment can create boundaries that the player's tank cannot cross (such as a ledge or a steep cliff wall.) The environment can also play a role in a player's defensive positions. The rugged terrain can rise steeply in front of a

user, providing cover from incoming fire. Any one environment can have a different layout of enemy and friendly units, and any single environment can be used in different missions.

While the exact size of an environment can vary from one environment to another, within the game they are generally large, seamless, and open. A player may not explore all of a single environment within a mission should they choose to not visit that location or be lead there by an objective. The environment can hint at the general direction of major areas with roads or recessed terrain, but in general there is no single linear path that a player follows within an environment.

2.2.1 Terrain

While the terrain may be multi-leveled at times, all units within the world will be participating on a single level plane of the environment. Deep ravines or tall mounds of earth will never be accessible for travel by any unit. The level on which a player participates may raise and lower slightly, but this has no effect on the player's ability to attack or move. The game play is conceptually two-dimensional in this way.

2.2.2 Environmental Objects

Environmental objects will sit on top of the terrain within the environment and can interact with the player. These objects provide cover and create a variety of battle experiences based on their layout in the environment. Environmental objects can be large and fixed in place, creating permanent defenses throughout the environment, or destructible, allowing a player to carve his own path through the world.

2.2.2.1 Fixed Objects

Environmental objects in the world, such as rocks and large structures, are intended to create obstacles or defenses for the player or non-player units to interact with. While all of these will make for impassable terrain, colliding with the tanks, some of them will block projectiles while others will not. None of these can be destroyed. Fixed objects can be useful for players to hide behind when attempting to avoid enemy fire.

2.2.2.2 Destructible Objects

Some objects within the world, such as trees, telephone poles, or small structures, can be destroyed by tank fire. These destructible objects will have predetermined amount of health before they are destroyed. Some objects, when destroyed, will disappear from the world, leaving only debris where they once stood, and no longer prohibit the tank from moving over them. Others, particularly larger objects such as buildings, may be destroyed but will not disappear entirely from the world and may still create an obstacle for the player. Destructible objects can serve as targets for mission objectives.

2.3 Events

Throughout most of a mission, the players will be engaging in free-form combat. The enemy units will be constantly engaging the enemy; friendly NPCs send troops at the enemy and it is up to the player to protect friendly units and destroy enemy units. However, to keep gameplay moving and provide a varied experience for the player, events are included with missions.

2.3.1 What an Event Is

Events are mini-skirmishes that have been pre-designed and do not rely on the enemy's overall strategy. Events will be more difficult and challenging than normal combat and will be recognizable due to the eclectic selection of enemy tanks attacking in higher numbers than normal or be composed of advanced specialty tanks.

Example I of an event: Ten small tanks come out of a forest and swarm around the players and pepper them with fire

Example II of an event: Having crossed the bridge, three large tanks attack from one side of the player while light tanks attack from the other.

2.3.2 What Causes an Event

Events will be triggered by a variety of player actions, including: arriving at a location, destroying a unit or building, or timer based. Triggering an event based on time is useful for defense maps where the players won't be moving around the map. Events should appear to be a mix between a random encounter and a response to a player doing something that catches the interest of the enemy.

2.3.3 Where Events Happen

When and where events take place depends on each mission. A mission will have a set number of events that a player can come across throughout a mission. To prevent the events from becoming predictable, there is a certain level of randomization unique to each mission dictating where the events will occur. Time since the last event will be taken into account when determining whether to trigger a pseudo-random event. If it has been a long time since the last event, the likelihood of an event happening will be higher.

3 Tanks

Tanks are the unit of choice in this game. Players control tanks, they shoot tanks, and they fight alongside tanks. While a turret may not conceptually be a tank, for the remainder of this document a turret will be considered a stationary tank.

Tanks are composed of two elements: the tread system and body and the turret. The treads allow a tank to move forward, backwards, turn, and rotate in place. The turret is where the gun goes and it rotates on top of the body allowing a tank to fire in any direction regardless of where the tank is heading.

3.1 Movement and Control

Tanks can be controlled by either a player or, in the case of non-player characters, by the computer. They move based on the same fundamental principles.

The control of a tank will be two-tiered. Controlling the base of the tank allows for turning and forward or reverse motion. Tanks can be rotated in place or while in forward or reverse motion. The second level of control over a tank is to control the tank's turret. The turret will be attached to the tank and will have the ability to rotate separately from the tank body's current rotation. This will allow tanks to avoid incoming enemy fire while at the same time returning fire.

Tanks are setup in the following manner:

- Tanks will be able to move forward and backwards at the same rate
- Tanks will accelerate and decelerate forward and backwards
- Tanks will be able to turn quickly while stationary
- Tanks will be able to turn while moving, but with both at a reduced rate
- Aiming will not be tied to movement or body orientation

Movement and Control Attributes

Maximum Speed	All tanks have a maximum movement speed. This maximum speed is reached after a period of acceleration. The maximum speed that a tank moves in is the same moving forward or in reverse.
Turn Speed	Turn speed defines the rate at which a vehicle is able to turn. There is no acceleration leading into turn speed, and a vehicle turns clockwise and counterclockwise at the same rate.
Turret Turn Speed	The turret turn speed is the rate at which a vehicle's turret turns towards the user's desired aiming direction. The turret does not accelerate into or out of its turning rotation. Turret turn speed also moves at the same rate clockwise and counter-clockwise. This speed

	will be higher than the tank's rotation rate. Note that the alternate machine gun on each tank will have no rotation speed and will automatically face where the player aims.
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Acceleration	Acceleration defines the speed at which tanks are able to change their speed. Heavier tanks accelerate at slower speeds than lighter tanks. Rate of acceleration is such that maximum speed is reached quickly. A tank's rate of deceleration (when changing between forward movement and backward movement or when coming to a stop) is based on its acceleration in that a tank decelerates significantly faster than it accelerates.
---------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 Shields

A tank has a limited ability to take damage before the tank is considered dead. This is defined by a tank's *shields*. When a tank is damaged by an enemy, this draws from the tank's *shields*. Although a tank may appear damaged as its shields lower, the tank will remain fully functional until all shields are gone.

Shield Attributes

Total Shields	The full potential shields that a tank has. This will vary from one tank to another.
Shield Type	Light or Heavy <ul style="list-style-type: none"> • Light armor is susceptible to machine gun fire (100%) • Heavy is nearly invulnerable to machine gun fire (10%)

3.3 Weapons

All tanks have a weapon; a large turret mounted barrel. From this, they will fire a variety of projectiles at varying speeds, rate of fire, range, and damage. These values depend on which tank is shooting.

NPC tanks will have one gun each and will fire at a prescribed rate.

Player tanks run off an energy system and have more variables describing them.

Please see the Player section for a more detailed description of how their weapons work and the NPC section on how patterns work.

3.3.1 Projectiles

After a tank shoots its gun at a location, the projectile will move towards the targeted location, hitting and damaging anything it comes across, on the way. If it does hit something, it explodes and does damage to that target. Thus, weapons fire will not fly over tanks or collidable units on the way to the targeted location.

Player shot projectiles will not overshoot the targeted location. If the projectile hits nothing, it will end up impacting on the ground and doing area of effect damage around the impact point. For normal player tanks, their projectiles will have a small amount of area effect damage around their explosion point. This damage falls off quickly so that the player will do very little damage to anything farther than a tank's length away from the explosion.

NPC projectiles will overshoot and their projectile will continue travel until they run out of range, at which point they land harmlessly doing no damage. They have no AOE damage.

3.3.2 Artillery

Artillery tanks function slightly differently than all the other weapons. Instead of working on a projectile based system that will hit any unit on its way to the target, it will instead take longer to arrive, but will impact directly at the targeted location in a circle around that point. This means an artillery tank can shoot over friendly and enemy tanks in order to fire upon a specific location, however it will not fire over map obstacles and will instead hit them. This allows tanks to hide behind walls, trees, etc in order to avoid AOE fire. This applies to both player and NPC artillery fire. For NPC artillery fire, a target icon will be painted on the ground shortly before the barrage impacts, giving the player time to get out of its way.

3.4 Friendly Fire

Friendly fire will exist in a restricted manner. NPCs can hit allied NPCs. Friendly NPCs cannot hit players. Players can hit each other with their primary weapon, but not their secondary, machine gun, weapon. Friendly NPC artillery can still hit players the same as enemy artillery.

This allows for as much friendly fire as possible, while preventing friendly NPCs from randomly killing players through their own incompetence. Friendly artillery can still damage players as their barrage can be dodged. If this ends up being problematic, solutions include: making it not hit players and removing friendly artillery.

4 Player

In the game, each player will be controlling a single tank. They will be fighting alongside fellow players and the friendly NPCs as they fight the enemy army, but his control is limited to his tank. The player will have no direct control over friendly or enemy units. The player's actions will, however, be taken into account and adjusted to by the non player character units. This is explained in the NPC section.

There are various qualities that define the way a player experiences driving a tank. These qualities can change between tank classes and aid in defining a tank's optimal play style and role in the battlefield.

4.1 Lives and Death

A player must survive in order to complete an objective within the game. A player's survival is governed by their tank's shields. Unlike NPC tanks, a player tank's shields regenerate over time. This regeneration rate is moderately rapid, limiting the "down time" that a player must spend waiting for their shields to return, but not fast enough to make them immune to battle. When a player tank's shields run out, their tank is rendered useless and the player dies. When a player's tank runs out of shields, however, the mission is not necessarily over. In order to create a gameplay experience that gives the player the opportunity to correct their mistakes and improve their tactics without starting over, they have a prescribed number of lives to use.

4.1.1 Lives

4.1.1.1 Single Player

The player has a specific number of lives to use in each mission. These lives apply to his tank alone, as opposed to the lives of friendly tanks. Running out of health within a mission will cause one of these lives to be used. When the player dies and has no lives remaining, the mission ends.

4.1.1.2 Multiplayer

In a multiplayer setting, the players participating in a single game share a pool of lives. There will be extra lives added to the pool of lives for each player in the game. For example, if the mission began with two players, they would have more lives between them to use in order to complete the mission than in single player.

It is established this way in order to encourage teamwork between players and prevent elimination. While this may facilitate bad behavior in an anonymous audience, where one player ruins the experience for the others by using all of the lives shared between players, it is not a major consideration in the scope of this game because the audience is expected to mainly play with friends.

4.1.2 Death and Rebirth

When a player dies and has a life available to respawn, they are momentarily removed from gameplay. This happens to punish the player for dying, give the enemy force an opportunity to regain its strength, and similarly give the player a period of time to understand their situation and strategize. They are brought back to life after this period of time at the last completed objective or, should no objectives

have been completed, the mission starting point. This is presumably a safe place for a player to re-engage in battle.

A player has no control over their vehicle during this time, after dying and before respawning. They may choose to view the map or bring up game options, but they are unable to interact with the world.

The following describes a sequence of events that take place when a player dies:

When a player dies, it is made apparent graphically through a hue applied to the screen and death text indicating that they have died. Their tank is clearly destroyed and disappears from the screen, leaving only fragmented remains.

After a short period, a player's camera is moved towards the location of their next spawn point. This allows the player the opportunity to understand their location in the world.

4.1.3 Death Without Rebirth

Should a player die without any lives remaining, the sequence starts similarly to death with rebirth, in that there is a clear indication of death through graphical cues and text. However, their camera will make no action towards a new spawn location. Instead, after the death sequence, the game fades to black and presents them with their game statistics.

4.1.3.1 Multiplayer

When a player dies and there are no remaining shared lives left, although team mates are still living, the death sequence will start off as it does in single player. Following this, the camera will snap to a random, living player's position in the game and follow exactly what their camera is experiencing. The "hue of death" will still exist for the dead player. Should the player they are viewing die, the player's camera will jump to a new player at the same time as the dead player's camera.

A player is also able to switch their follow-camera through their teammates. They will switch through the players that they are able to view as the players are listed down the side of the users screen. They are able to switch forwards and backwards through the players that they wish to view.

When the last player dies, everyone will experience the fade out at the same time and be brought to mission statistics after the game.

4.2 Attack and Damage

A player's tank has two weapons that it is able to utilize in battle: the primary cannon and the secondary machine gun. A player is able to fire their primary cannon when they have the required energy to do so. A player's primary cannon will be limited not only by its rate of fire, but also by the energy available. If the tank does not have enough energy, the gun will not fire. The secondary machine gun, while retaining the limiting rate of fire, does not use energy and is not limited by the energy pool.

Attack and Damage Attributes

Primary Damage	How much damage the receiving tank will get
Primary Attack Range	How far the projectile will travel before exploding on the ground
Primary Firing Rate	How much time must pass in between shots
Energy Cost	What percentage of energy the shot requires and how much it will decrement the energy pool
Area of Effect	How big of an area around the shot will take damage
Area of Effect Drop-off	The rate at which damage is reduced based on distance
Projectile Speed	How fast the projectile will travel, will affect how much the player should lead and how easy it will be to dodge

4.2.1 Primary Weapon

When a player fires their *primary weapon*, a projectile will fly from the barrel of their cannon towards the targeted location. This projectile travels out of the barrel in the direction of the turret at a speed which still allows for some tanks to out maneuver the projectile. A projectile will *not necessarily* move exactly where the player is attempting to fire, as the turret may be moving towards where the player is aiming. If a player's turret has not caught up to the position they are aiming for, the projectile will fire from the position the turret is in. The projectile will collide with any object it contacts and will not fly over objects on its way to the targeted location.

Where a player is able to fire is governed by the primary attack range, which describes how far a tank's shell can fire and differs for each tank class. If the player aims at a position out of range of the tank, the reticle will change color and the projectile will land at the farthest point it can reach. When the projectile hits a target, or reaches the targeted location, it will explode causing damage in a small radius around it. This damage will fall off quickly as the distance from the center of the explosion increases. This will allow a player to more effectively destroy densely grouped enemies by firing into the middle of them. The Lancer is unique in that its radius is much larger and the damage doesn't drop off nearly as quickly, effectively allowing it to hit a number of enemies with full force.

4.2.2 Secondary Weapon

In addition to their primary gun, player tanks will have a secondary machine gun. The machine gun does less damage than the primary gun, but at a higher rate of fire and does not use energy. This makes it exceptional against lightly armored enemies, but less useful against heavily armored foes. It also benefits from not having a maximum turn rate and will always fire wherever the player is aiming. This gun also differs from the primary weapon in that one cannot dodge the projectiles it fires because they move nearly instantaneously.

A player's machine gun is situated on top of the rotating tank body. The machine gun, however, turns instantly to where the player is trying to aim. This helps the player quickly target enemies that may come up behind them when their cannon is not able to turn quickly enough towards the targeted location.

4.2.3 Minimum Attack Range

For both the primary and secondary weapons on a tank, there is a minimum attack range. This is the shortest distance that a tank is able to fire, and prevents a tank from shooting itself. This range is just beyond the length of the tank. If a player attempts to position their aiming reticule within the minimum attack range, it colors itself in a manner that lets the player know that they are unable to hit that exact spot. Firing while targeting a spot within the minimum attack range will send the projectile in the direction of the turret or gun and hit the nearest possible location to that spot without causing damage to the player's vehicle.

4.2.4 Energy

Energy is the resource needed in order to fire a player's primary weapon. While all player tanks will have the same amount of energy available, different guns will consume different amounts of energy and different tanks will have differing regeneration rates. For example: one tank may drain its energy in 2 shots, but then regenerate all of its energy in 3 seconds, while another tank may be able to fire 6 shots before its energy is depleted, but will take 10 seconds to regenerate to full.

Energy Attributes

Energy Regeneration Rate	How fast the main weapon energy regenerates
--------------------------	---------------------------------------------

4.3 Player Tank Classes

Players will have the option to choose from four different tank classes in order to increase replay-ability as well as add depth to the multiplayer experience. The goal is that all four tanks are equally effective, but with their own strengths and weaknesses which promotes different play styles. The classes will complement one another such that two players with different tanks will be more than twice as effective when working together than alone, encouraging cooperation. In the previous section, various attributes were described that define how a single tank operates.

Several things can change from one tank to another including: physical size, speed, acceleration, turn rate, rate of fire (for the primary gun), amount of shields, primary gun damage, and range. There are also less easily quantifiable changes like increasing the tank projectile's area of effect. Increasing this would be effective against large groups of enemies but coupled with a low rate of fire would be bad against single targets.

All player tanks have steadily recharging shields and weapon energy. Shield health varies from 4-10 shots depending on the player tank. Small enemies can be taken down with a few machine gun bullets, or one primary gun. Larger enemies will take several primary shots or many, many machine gun bullets.

One of the primary ways of making the experience of playing as each tank unique, is the varying of energy usage. By giving one tank a weapon that drains energy quickly, but also regenerates energy quickly, a tank that has a relatively steady rate of fire is created. By making a tank that can fire many shots before it runs out of energy and then giving it a very slow regeneration rate, a tank that has great “burst damage” potential but needs to rest between battles is created. This, coupled with appropriate shields and speed, defines the styles for the player tank classes.

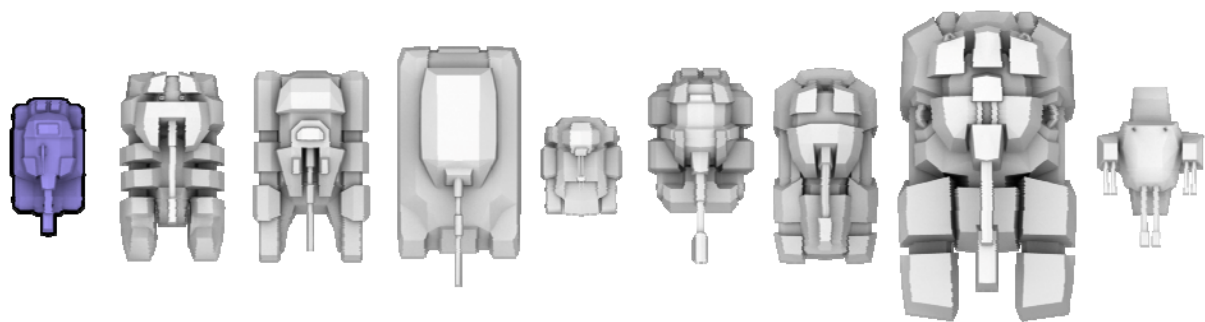
4.3.1 “The Hornet”



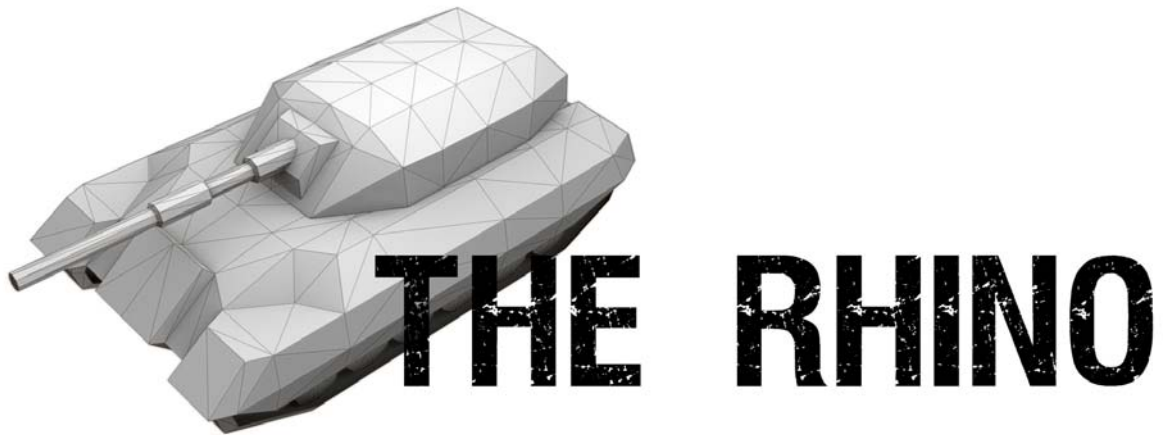
THE HORNET

The Hornet tank is more like a heavily armored off-road vehicle with a heavy caliber weapon than a tank. It will excel at battles against lots of small enemies with its fast firing light cannon. Due to its low armor, it will need to hop in and out of battle to let its shields recharge.

Size	5 meters in length
Movement Speed	Fast
Turn Speed	Fast
Turret Turn Speed	Fast
Acceleration	Fast
Total Shields	Low
Shield Regeneration Rate	Medium
Primary Damage	Medium
Primary Attack Range	Low (1/2th of screen distance)
Primary Firing Rate	Fast
Primary Energy Cost	High
Area of Effect	Very Low
Projectile Speed	Fast
Energy Regeneration Rate	Fast
Extra Notes	Energy configuration results in a high overall rate of fire.

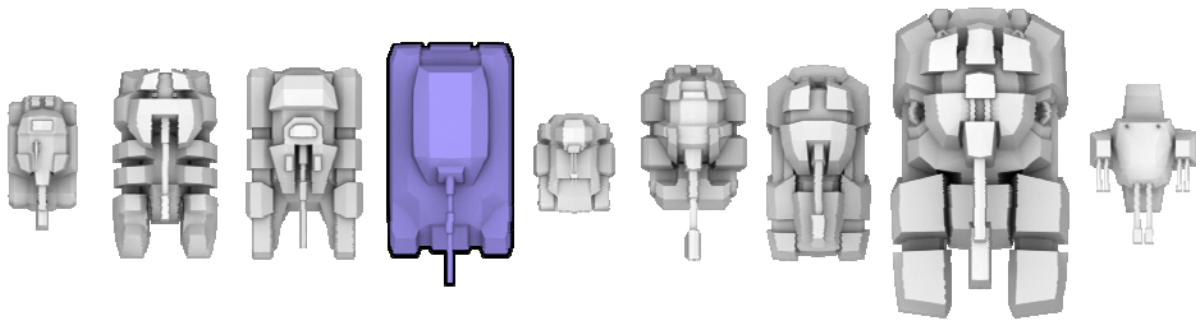


4.3.2 “The Rhino”

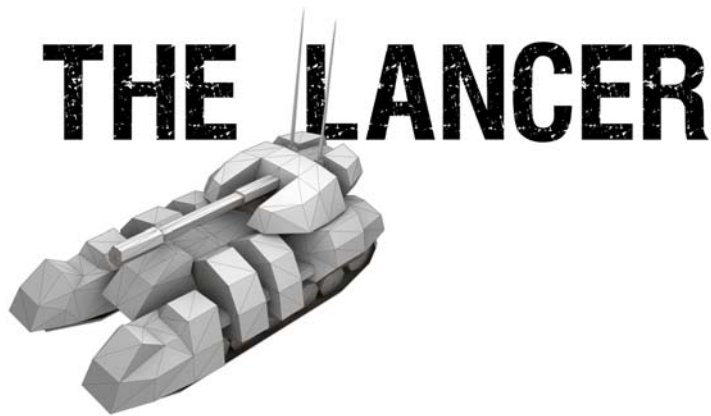


The Rhino is more of a traditional tank; roll in, set up camp and begin blasting everything in sight. With its recharging shields, it can take a lot of punishment as long as it is not moving around too much. A strong tank that is good for a traditional play style.

Size	12 Meters in length
Movement Speed	Slow
Turn Speed	Medium
Turret Turn Speed	Medium
Acceleration	Medium
Total Shields	High
Shield Regeneration Rate	High
Primary Damage	High
Primary Attack Range	High (Screen distance)
Primary Firing Rate	Medium
Energy Cost	Medium
Area of Effect	Small
Projectile Speed	Medium
Energy Regeneration Rate	Medium



4.3.3 “The Lancer”

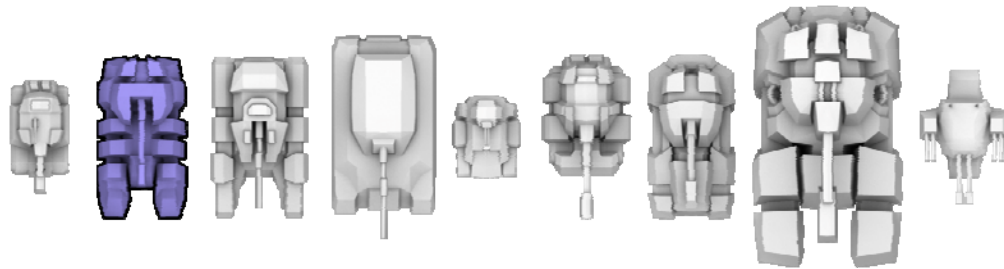


The Lancer is an artillery tank, although it is much more mobile than traditional artillery. It can move while firing and its longer range explosive rounds are excellent for taking down groups of enemy units of all strengths. It must remain stationary in order to recharge its guns, limiting its mobility. This tank prefers to hide behind its allies.

Size	8 meters in length
Movement Speed	Slow
Turn Speed	Slow
Turret Turn Speed	Slow
Acceleration	Slow
Total Shields	Low
Shield Regeneration Rate	Slow
Primary Damage	High
Primary Attack Range	Very High (More than screen distance)
Primary Firing Rate	Slow
Energy Cost	High
Area of Effect	Large
Projectile Speed	Very Slow
Energy Regeneration Rate	Medium
Extra Notes	The Lancer can only regenerate weapon energy when stationary.

This tank will have the ability to zoom-out, allowing it to see and shoot farther than all the other tanks. (See the Camera section for more info)

Lancer projectiles do not impact objects they come across and will always explode at the targeted location.



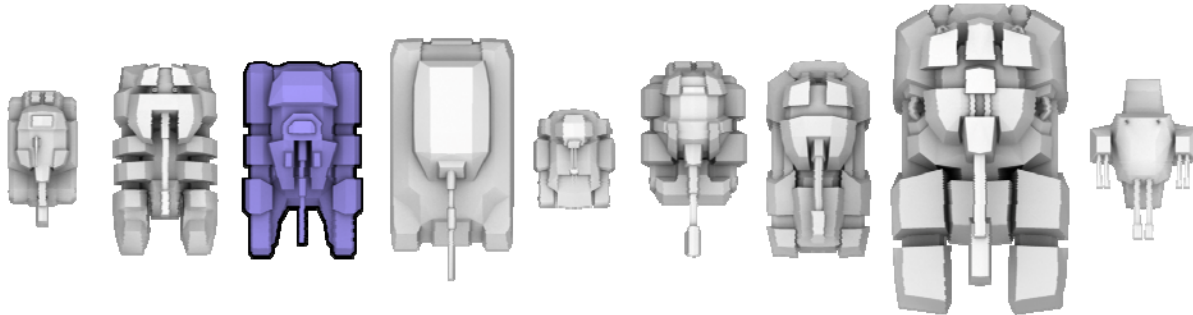
4.3.4 “The Ant”



The Ant is a specialized hit and run tank. Its weapon is designed to be used in short bursts, unleashing a torrent and then backing off to recharge. A skillful Ant driver will make use of its regenerative shields that supply its cannon with extra energy when attacked to extend its barrage before it must retreat.

Size	8 meters in length
Movement Speed	Medium
Turn Speed	High
Turret Turn Speed	Medium
Acceleration	Medium
Total Shields	Medium
Shield Regeneration Rate	High
Primary Damage	Medium
Primary Attack Range	Medium (2/3 rd of screen distance)
Primary Firing Rate	High
Energy Cost	Low
Area of Effect	Small

Projectile Speed	High
Energy Regeneration Rate	Medium
Extra Notes	The Ant has a large amount of energy (enough to fire 10 shots without recharging) but this energy regenerates faster when the Ant is taking fire.



5 Non-Player Characters

During gameplay, the player will encounter units that are on the same team as the player. These units will assist the player in the battle to help complete their objectives. These units will constantly be on the battle field and progress up the battlefield as the player progresses through various objectives. The player will also encounter units that are not on their team. These units will be against the player trying to stop them from progressing through the mission. The units on the opposite team of the player will fight both the player controlled units and the non player controlled units.

This section describes, in detail, each of the units that the player will encounter on the battlefield. Each unit has a variety of abilities and personality attributes that can work with or against the player in their progress.

5.1 Spawning

While there are a number of both friendly and enemy NPCs populating the map at the start of a mission, these eventually die and need to be reinforced. This is done through the regular spawning of units. The spawning of units is not tied to deaths, but instead purely based on time. Every so often, a group of units will be spawned. Which units and how many are spawned is determined by the mission. Different groups of units can be set to spawn at different rates, allowing for alternating group make-ups. Units spawn at certain locations, specified in the map description. There is no observable limit to the number of units that can spawn. The number of NPCs is balanced by the spawn rate and the death rate. The spawn rate will be constant over the course of the game.

5.2 Firing

NPC tanks function slightly differently from the player when it comes to firing. While players are limited by their energy, NPCs are limited only by their maximum rate of fire and will not be limited by energy. Enemy fixed turrets will use a firing pattern when shooting, to make their behavior slightly less monotonous. An example of a firing pattern is three short shots followed by a pause and then three more shots.

5.3 Friendly Units

The units that fight on the player side are on the battlefield from the moment the player arrives on the scene. These units will progress up the map to assist the player in completing their objectives along with the mission as a whole.

Reinforcements will arrive periodically from allied bases located near the player's starting location. The bases will produce a specific group of units. Each group will take time to travel from their base to the battlefield. As they arrive on the battlefield, these units will join the battle and fill in the gaps where they are called for. The player has no control over where these new units engage in battle or what types of units are sent to assist the battle.

Friendly units will push the front line forward and progress to the closest objective if they are given the chance. These units will attempt to accomplish objectives, however often they will not be able to do this feat alone. While friendly tanks work together, they rely on the assistance of the player to guide the way and turn the tide of the battle.

If the player neglects assisting his friendly non player controlled units, these units may not succeed in progressing across the map. The friendly units can be easily overcome by enemy units and lose ground without the assistance of the player. It is important for a player to help these units keep ground and progress along the mission at hand.

5.4 Enemy Units

The units that fight against the player are enemy units. These units play on the opposite team of the player and are not controlled. The actions of the player will however affect the actions of the enemy units.

Similar to the player's friendly units, the enemy units will also originate from various bases located outside the battlefield. As with the friendly units, these units will take time to travel from their home base to the battlefield. As they arrive on the battlefield they will progress to the front line and provide backup for places that may be weak.

Enemy units may be patrolling an area of the map. These units will report sightings of the player's team to the rest of their army. This reporting may trigger other patrolling tanks to assist in stopping the player's progress. This may also trigger tanks that are in an idle position defending an objective, or structure nearby.

Other enemy units may be idle defending structures, or objectives around the map. These tanks follow the same reporting system as the patrolling tanks. These tanks send out an alert that may activate other idle tanks. These tanks are often dangerous to stumble upon for the player as idle tanks often are found in groups guarding a specific area. When the player encounters these units they should expect reinforcements be close by.

A particular unit the enemy has that the player's side does not is the turret. Turrets are a defense mechanism that does not move. They have specific firing patterns and will aim at friendly units including the player.

5.5 NPC Behavior

Units on both teams will exhibit similar behaviors. These behaviors are broken down into two levels. The team will have its own strategy and the individuals will attempt to execute that strategy. The individual behaviors will change depending on the type of unit that is on the battle field.

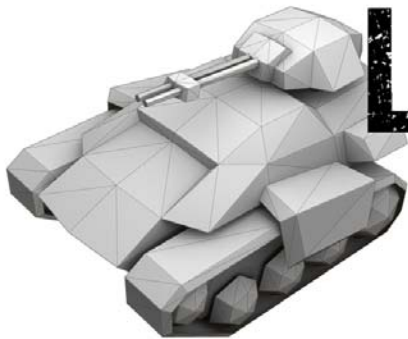
Each type of unit will have a specific personality described in detail below; this personality will determine the way the unit will fight. The units will exhibit their behavior when entering combat as well as when wandering around the battlefield looking for the opposing team.

The team level behavior the units will follow will attempt to maximize the effectiveness of that team. For the player's team, this means that the team will attempt to progress up the battlefield and complete objectives. On the opposing side, this means the units will defend objectives and attempt to stop the player from progressing. The units that have these goals are ones that arrive as reinforcements, not units that begin the game already deployed.

Units that begin deployed to the battlefield will have localized objectives. These objectives will be to defend a specific point or to patrol between several points. These units will not follow the team level behavior and instead, these units will follow the roles they have been assigned. The player's team will not have units that follow this behavior.

5.6 NPC Units

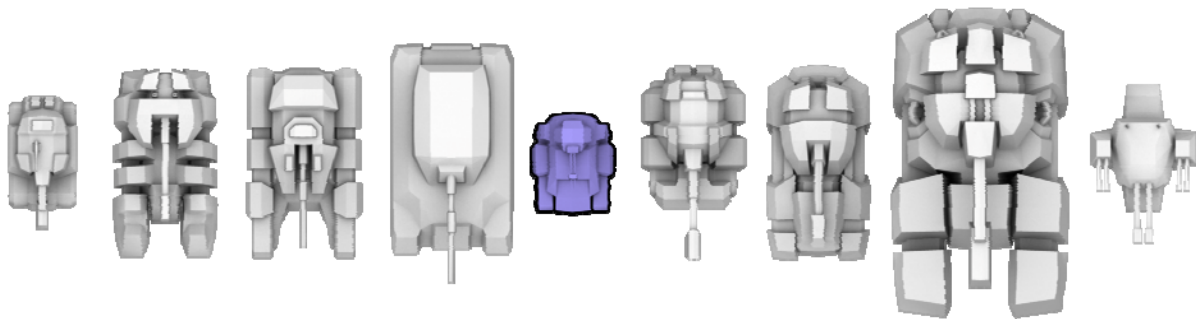
5.6.1 Light Tank (Wolfhound)



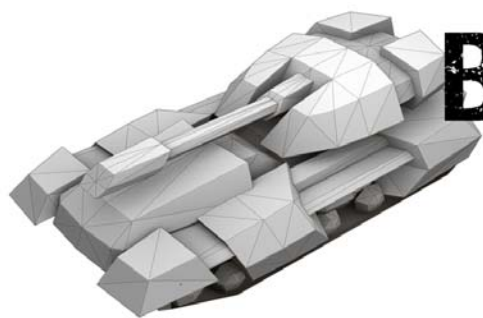
LIGHT TANK

These are a class of fast, light units. They come in two varieties with different behaviors.

Size	4 meters in length
Movement Speed	Fast
Turn Speed	Fast
Turret Turn Speed	Fast
Acceleration	High
Total Shields	Low
Primary Damage	Low
Primary Attack Range	Short (1/4rd of screen distance)
Primary Firing Rate	High
Area of Effect	None
Projectile Speed	High
Behavior Type-A (Pack)	Tend to seek groups of friendly units, especially other Wolfhounds. This unit prefers to attacks in groups using hit-and-run tactics. Often position themselves with other tanks in the area to flank the enemy. Move in unorganized flocks for speed.
Behavior Type-B (Scout)	Advance into enemy territory in order to gather intelligence about the enemy. This unit will run into an enemy crowd to gather numbers and information about the enemy positions. This unit will then proceed to run when spotted. This unit has the ability to fire but only does so if the situation absolutely requires it – if firing will help the tank escape and not draw any more enemy units.



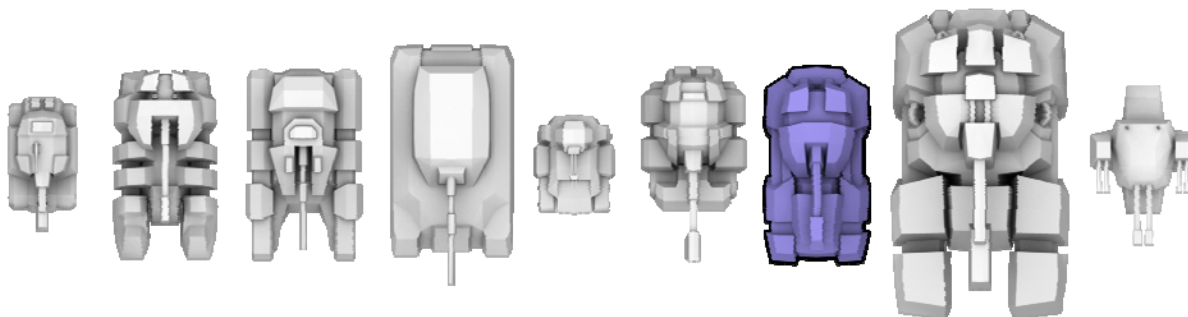
5.6.2 Battle Tank (Durant)



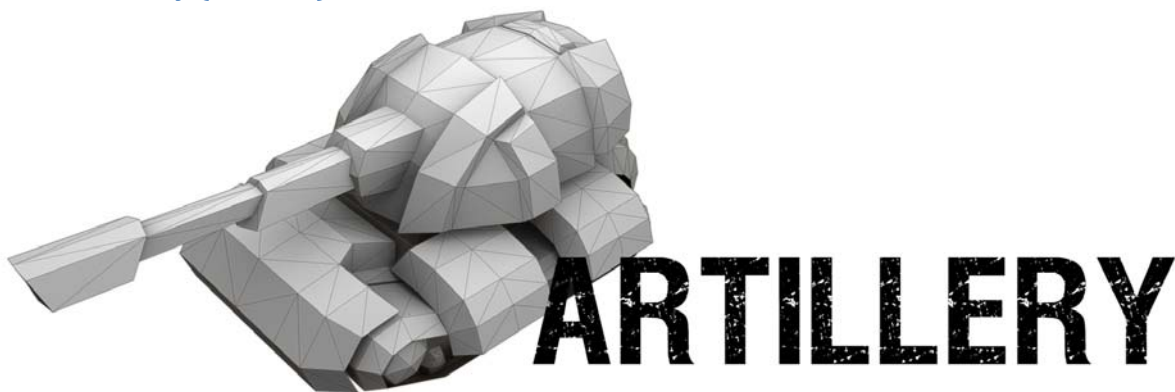
BATTLE TANK

These are medium units in speed, armor and firepower. They are the front line units, and used as support for players and heavier units. The variety of behaviors for this unit allows squads to show some variation. These tend to move in formation when moving in groups.

Size	8 meters in length
Movement Speed	Medium
Turn Speed	Medium
Turret Turn Speed	Medium
Acceleration	Medium
Total Shields	Medium
Primary Damage	Medium
Primary Attack Range	Medium (2/3rd of screen size)
Primary Firing Rate	Medium-High
Area of Effect	None
Projectile Speed	Medium
Behavior Type-A (Battalion)	This behavior calls for a defensive strategy. This unit will move directly to objectives and take a hide and seek approach to surprising the enemy. When this unit is required to move from its current position it will often move in a formation with other units of the same type.
Behavior Type-B (Maverick)	This unit will charge into battle in hope of taking down a few units then moving out of range of the remaining units to reload, before going back into battle.
Behavior Type-D (Duck and Cover)	This unit likes to hide behind cover or friendly units, and only shows itself to fire a few rounds and returns.

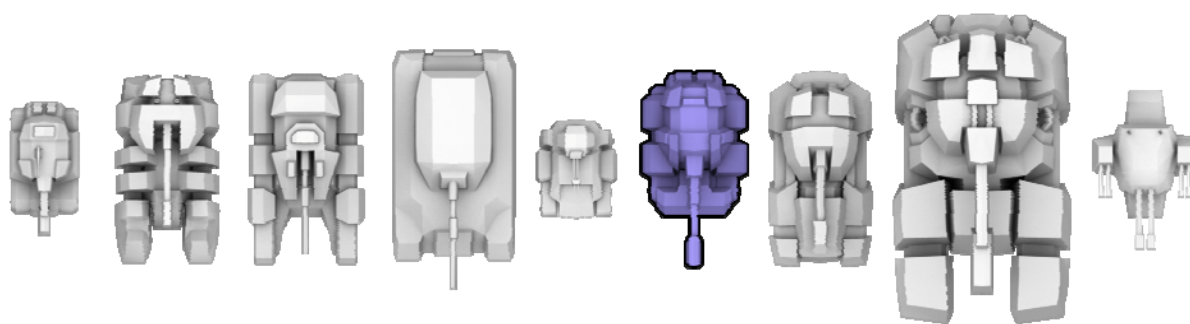


5.6.3 Artillery (BRAMM)



The Ballistic Round and Missile Mobile (BRAMM) units are used more as heavy artillery, firing highly explosive rounds rather than dumb shells. While these rounds do massive damage and travel over units, the BRAMM itself has relatively low armor and takes a while to deploy. The main behavior for this unit is to park itself in a secluded area from which it can bombard enemy units.

Size	8 meters in length
Movement Speed	Medium
Turn Speed	Medium
Turret Turn Speed	Slow
Acceleration	Medium
Total Shields	Medium
Primary Damage	High
Primary Attack Range	High (Entire screen)
Primary Firing Rate	Low
Area of Effect	High
Projectile Speed	Medium
Behavior Type-A	<p>In order to attack, this unit is required to stop and deploy first, making a distinctive noise warning players of its presence. After this unit is fully deployed, it then can target units in a large radius around this unit and fire at them. With a slow fire rate but high damage and area of effect, this unit likes to sit just behind the frontline and target groups of enemies. This unit is vulnerable to units that are close to it, and is slow to un-deploy before it can move again.</p> <p>When firing, a large circular target image will show up on the ground that is being targeted, alerting players and giving them a chance to clear out before the barrage hits.</p>

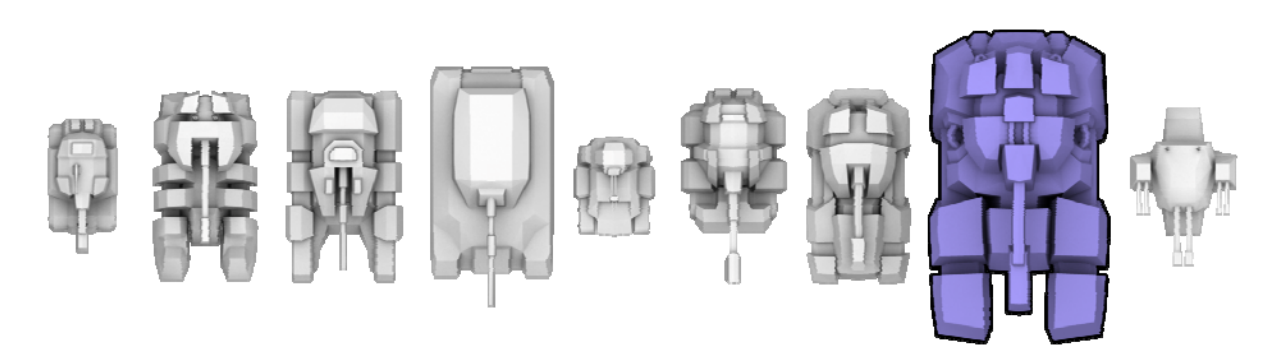


5.6.4 Heavy Tank (Stonewall)

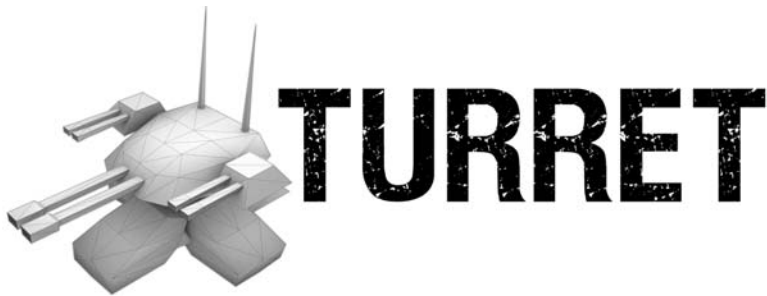


These behemoths of the battlefield are a force to be feared. Only showing up during the direst of situations, they are slow, heavily armored, and pack a powerful punch. Often accompanied by a large escort, these should be taken care of quickly. Two or three Heavy Tanks in one place is a rare sight.

Size	16 meters in length
Movement Speed	Slow
Turn Speed	Slow
Turret Turn Speed	Slow
Acceleration	Slow
Total Shields	Very High
Primary Damage	High
Primary Attack Range	High (Entire screen)
Primary Firing Rate	Low
Area of Effect	Low
Projectile Speed	Medium
Behavior Type-A	This unit is a leader, often associating itself with group seeing units and leading them into battle. Known for charging into the middle of a fray and being the only one to survive.

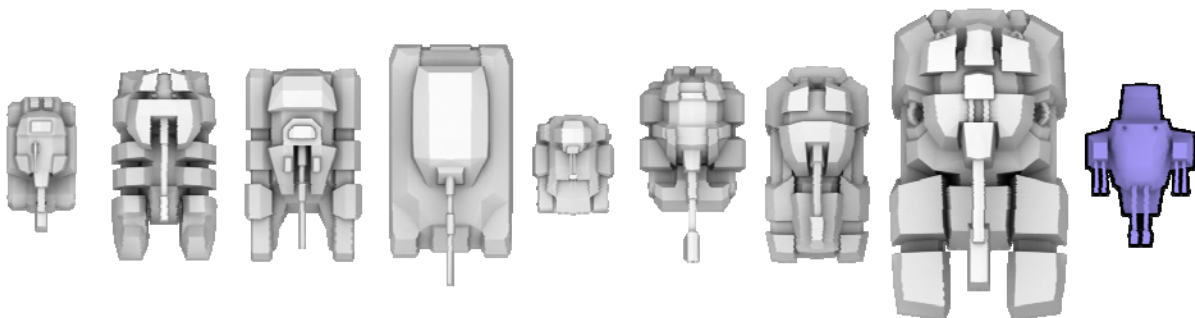


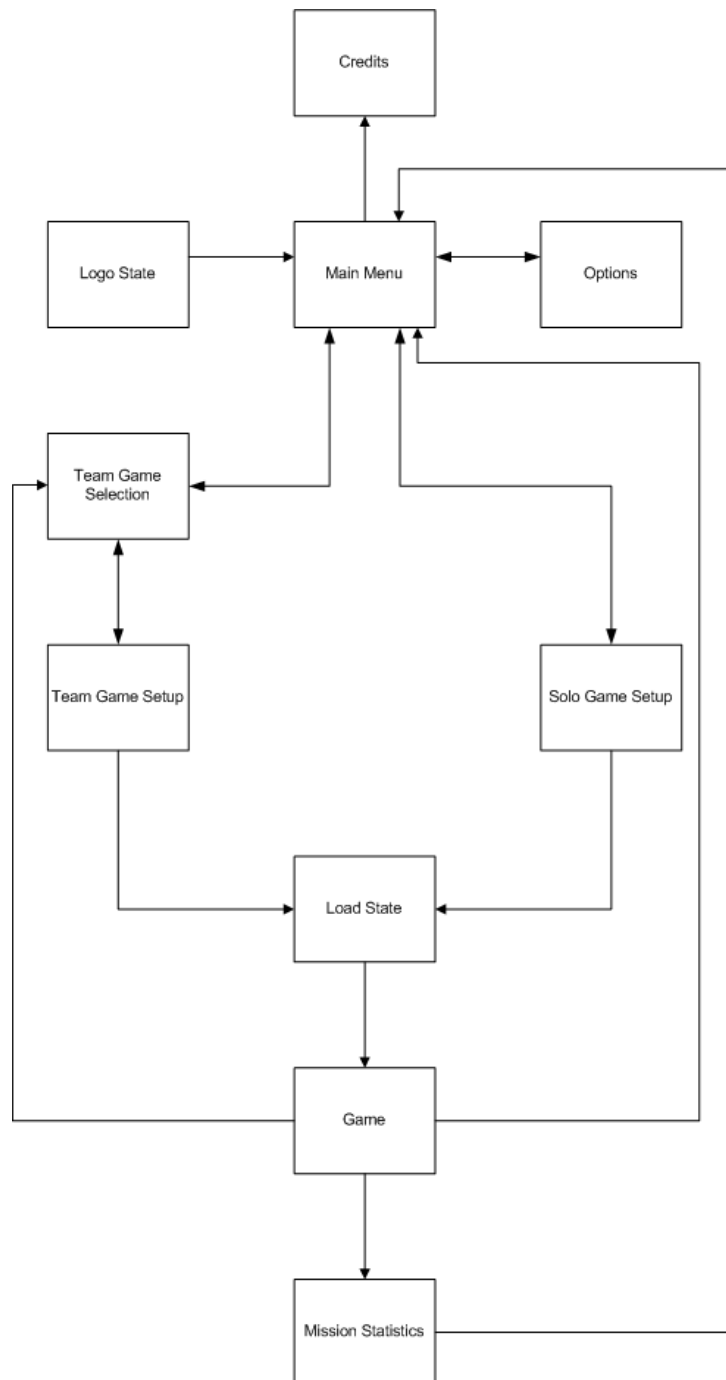
5.6.5 Turrets



Turrets are stationary enemy units that rotate and fire at opposing targets within their range. Turrets will never respawn during a mission. Turrets can be found throughout the map surrounding heavily-guarded areas or near a mission objective. There are two types of stationary turrets available: Light Machine Gun placements and Heavy Artillery. All turrets have the same basic behavior: shoot at any enemy targets within range.

Size	4 meters in length	
Movement Speed	None	
Turn Speed	None	
Turret Turn Speed	Medium	
Acceleration	None	
Total Shields	Low	High
Primary Damage	Low	High
Primary Attack Range	Low	High
Primary Firing Rate	Low	High
Area of Effect	None	High
Projectile Speed	Medium	High
Behavior Type-A	This unit will fire at the first unit in range.	
Behavior Type-B	This unit will fire at units in range that have low health	
Behavior Type-C	This unit will fire at the closet unit.	
Comments	Regardless of the behavior, this type of unit has various firing patterns. The firing patterns can be applied to each type of behavior.	





6 Game States / Flow

The different states of the game and their transitions should not only provide a clean, usable experience, but should reflect the modern style of the game. Moving between screens should be quick with subtle animations and movement of user interface elements.

6.1 Logo Screen

The logo screen will be used to briefly promote the RIT games program and mention that the game is developed by the game graduate students of 2009. It will be easily skipped with user input, but should also go automatically to the menu after a set amount of time.

6.2 Menu Screen

The menu screen will fade in quickly after the logo screen or when coming from another state. Animation into the menu screen will subtly slide user interface elements into place. The transition itself will be very quick, allowing the user to experience the immersive transition but not creating an undesirable experience. The first time the menu screen is loaded, whether coming from the logo screen or the game, it will introduce the menu music. Entering the logo screen from another state will transition into the state more quickly than when it is transitioned to from the logo state. When coming from another game state, the transition into the menu state will be almost immediate. This aims to avoid annoying the user with a transition each time.

From the menu screen, the user is able to do the following: start a solo mission, start a team mission, access options, view the game credits, and exit the game.

6.2.1 Start Solo Mission

Starting a solo mission brings the player to the *Solo Game Setup* room. The menu screen will fade to black shortly and user interface elements will subtly move to the side.

6.2.2 Start Team Mission

Starting a team mission will present a dialogue box allowing the user to enter a multiplayer alias. When this dialogue box appears, the user interface will blur and darken in the background. A random name will be provided randomly from a list of default player names. Cursor focus will be on the dialogue box and should the user begin typing a name, the default provided player name will automatically disappear. The blur and darkening of the dialogue box will be almost instantaneous. If the user enters a name, or uses the default, and continues, they are then brought to the *Team Game Selection* room. If a user chooses to return to the menu, the dialogue box will disappear and the game state will reappear again.

6.2.3 Game Options

In this state, the user has the ability to change a number of game options. Transitioning out of this state will slightly animate the menu state objects out of the way and quickly fade to black.

6.2.4 Credits

The player is able to view the credits from the menu state without needing to exit the game. Clicking on the credits option will transition the menu state out and fade the credits in as they would be faded in when the player exits the game.

6.2.5 Exit

When a user exits the game, they are taken to the credits screen. Transitioning from the menu state to the credits screen is slightly different than the other transitions. The menu screen will not animate on the exiting of the game, but should fade to black over a short period. It will be possible to skip this screen and immediately exit.

6.3 Game Options

The game options screen presents the player with options regarding the games performance and controls.

6.3.1 Graphics

Graphics options will allow the toggling on and off of certain graphical features and changing levels of other graphical features. A user will also be able to modify the games resolution and full screen settings here.

6.3.2 Controls

The controls screen will allow a user to change any controls of the game.

- Tank Controls
 - Forward
 - Back
 - Turn Left
 - Turn Right
 - Fire
 - Alternate Fire
- Misc
 - Pause
 - Screenshot
 - Open Console
 - Open chat window

6.4 Solo Game Setup

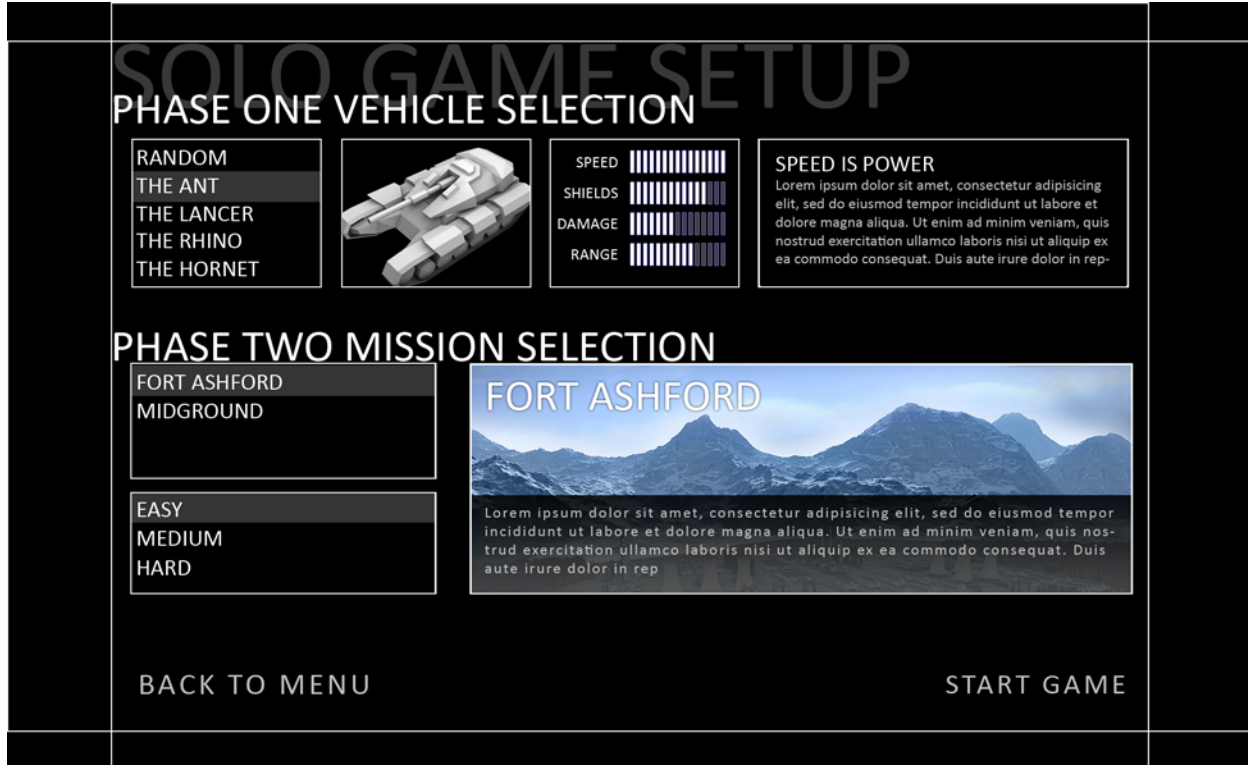


Figure 7.4-1 A conceptual mock up of the layout of the Solo Game Setup screen

The Solo Game Setup screen allows the user to select their player class, mission, and difficulty. All options are set within a single screen. A user can exit from the screen to return to the main menu or continue on to the Mission Info / Game Loading screen.

6.4.1 Player Class

When selecting a tank, the player has five choices: Ant, Hornet, Lancer, Rhino, and Random. By default, the selection begins with a random indicator. If a player starts the game with “random”, the player’s class selection will be randomly selected after the player starts the game. Player class information should be displayed through bars. The selection of a player class should not cycle around once the last player class is reached, but instead should stop, in order to prevent confusion with the number of player classes available. Player class information should be presented with an image of the class, an overview of its statistics, and a brief description of the class.

Information that will be displayed for each tank includes:

- **Speed:** The speed of the tank class, which is a function of its max speed, acceleration, and turn rate.
- **Shields:** The tank class’s shields, which are a function of their max shields and regeneration rate.

- **Damage:** The damage value of a tank class. This is a function of its damage per shot and its rate of fire.
- **Range:** The maximum firing range of each tank.

6.4.2 Mission

Cycling through the missions should contain a thumbnail of the mission, a mission title, and a brief, catchy description of the mission. This will always default to the first mission in the missions list. Mission pictures will cross-fade as the selected mission changes.

6.4.3 Difficulty

Game difficulty can be one of any of the possible game difficulties. It should default to *Normal* difficulty. Other options will include easy and hard.

6.5 Team Game Selection

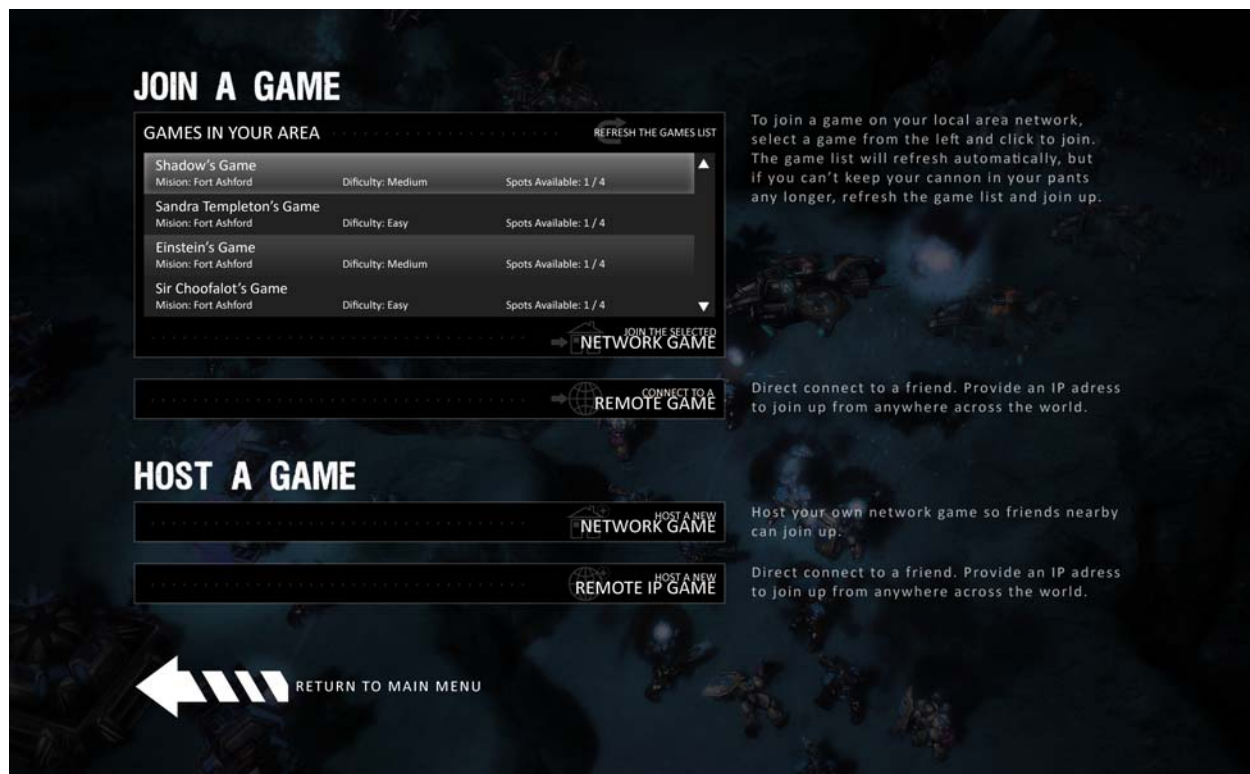


Figure 7.5-1 A conceptual mock up of the team game selection screen

The team game selection screen allows a player to select a game to play from a list of available multiplayer games, join a specific game based on IP, or create a game himself. This screen will populate itself with games hosted by other people should they be discovered. It will also allow a user to directly connect to a remote game. A user should be able to host any game type from this screen or join a remote game with a computer address. This screen should also display the user's alias for their reference.

The list of games should update itself periodically. It should also allow for a manual refresh from the user. Refreshing the list of available games will clear the list of games previously listed and quickly refresh them with a new list of games.

6.5.1 Game Names

A game's name should be a combination of the user's alias and text describing that it is their game room. For example, a game hosted by a user named "Renegade" should show up as "Renegade's Game."

6.5.2 Creating a Game

When the player creates a game, they are taken directly to the Team Game Setup room.

6.5.3 Joining a Game

When a player joins a game, the team game selection room will blur behind them and a connecting indicator should rapidly appear. If the game can be connected to, the connection icon should rapidly disappear and they should be brought to the Team Game Setup room. If the game cannot be connected to, the connection icon will rapidly disappear and the user will be presented with a dialogue informing them that they were unable to connect to the game. The dialogue box will have a button allowing the user to continue and pressing this button will cause the dialogue box to disappear and the team game selection room to reappear behind the dialogue box. Each transition should be a fraction of a second.

6.6 Team Game Setup

The team game setup room configures the options for a game before the game begins. The player that created a game is defined as the team leader, and is responsible for configuring and launching the game.

6.6.1 Configuring the Game

Similar to the Solo Game Setup room, the Team Game Setup room allows players to set up which tank class, map, and difficulty they would like the mission to be. In the Team Game Setup room, the host is responsible for selecting the map, the difficulty, and their own tank class. Any changes that the host makes regarding the map and the difficulty are visible to other players in the room. Other players that are not the host are responsible for selecting their own tank class. Any changes or choices that a player makes in the Team Game Setup room are visible to all other players.

6.6.2 Starting the Game

The host is the only player able to start a game in the Team Game Setup room. In order to do so, all players must have indicated that they are ready and agree to the settings that the host has selected. Each non-host player will have an icon to indicate that they are ready to launch the game, and when they are satisfied with their own tank selection and the host's game configuration, the host is able to launch the game. If a player decides to change their tank after they have been marked ready they are able to do so but they get changed to not ready. Launching the game will quickly fade the team game selection room out over the course of one second and lead them into the Mission Info / Game Loading screen.

6.6.3 Chat Box

The chat box allows users that are in the Team Game Setup room to communicate with one another and to get messages from the host when users are connecting and disconnecting from the game. A message is presented first with the alias of the user that sent the message and then their message.

Example

```
[Shadow]: Who is ready for some tank carnage??
```

Server messages also appear in the chat box. If a user connects or disconnects from the room, it will be presented as a server message.

Example

```
The player Shadow has left the game room.
```

6.7 Mission Info / Game Loading

The mission info / game loading screen is an opportunity for the game to load its game assets and present information on the mission that is about to be played. The information in the mission should consume the screen and there will also be a series of loading bars at the bottom of the screen informing the user of their loading progress. Individual bars will show progress for texture loading, model loading, and other content. Once the loading of the game is complete, the screen will fade out and the player is brought to their mission.

6.7.1 Team Game Considerations

In a team game, the mission info / game loading screen will wait for players to finish loading before continuing into the game. Along with the loading bar of the user, there will be information regarding whether or not other members of the team have finished loading the game. Once all members have completed the loading process, the game will begin.

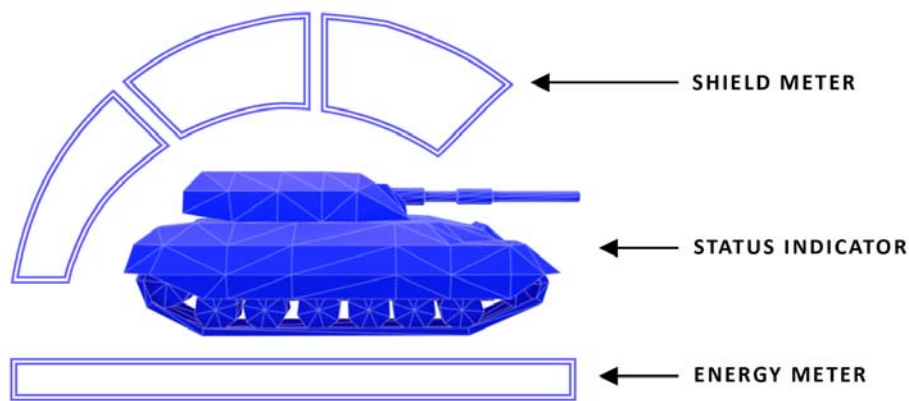
6.8 Game

From the game, there are two possible scenarios that can lead a player out of the game state. Should the player complete or fail the mission, they are brought to the mission statistics state, where statistics about the mission are presented to them. Also, from within the game, the player is able to exit through a menu, which will bring them directly to the main menu.

There are two important user interface elements to the game state: the Heads-Up Display and the Game Menu.

6.8.1 Status Panel

The Status Panel is part of the heads-up display within the game. A player's Status Panel provides feedback to the player regarding the condition of their tank.



The Status Panel will reflect the player tank's Shields and Energy through the following Heads-Up Display elements:

- Status Indicator
- Energy Bar
- Shield Meter

In a single player game, in order to view the status of their tank, a player will have a single Status Panel positioned in the upper left hand corner of their screen. In a multiplayer game, a player will also be able to see the status of other teammates in order to make better decisions based on their condition. This information is presented in smaller versions of the Status Panel aligned below the player's Status Panel. There will be one status panel for each player in a multiplayer game. A teammate's Status Panel is smaller than a player's in order to draw focus to the player's own panel and avoid cluttering the screen.

6.8.1.1 Status Indicator

The Status indicator is the central focus of a player's status panel. The status indicator should reflect the player's choice of tank class and provide general feedback on the condition of their tank. The Status Indicator should color itself from blue to grey depending on the player's health. This gives a general

feedback to the user that they can read from their periphery regarding the status of their tank without actually referring to their tank shield bar.

When a player is damaged, their Status Indicator should flash white quickly and fade back to its normal color. This provides the user with more subtle feedback when they are damaged. When a player is dead, the Status Indicator should slowly pulse from a dark red to a red, in order to give the player more indication of the fact that they have died. In a multiplayer situation, this will also allow players to see whether or not their allies are alive should they not be within visible range of one another.

6.8.1.2 Shield Meter

The shield meter is an indication of a player's shields in the game. The shield meter wraps around the top of the player's Status Indicator. The player's Shield Meter should be wide and informative, providing indicators along the meter showing where 2/3rd and 1/3rd levels of shield energy are. In a multiplayer situation, an allies' Shield Meter should be narrower and simpler, providing less detail to the user while consuming less of the screen.

The Shield Meter can be in one of four states: *steady*, *recharging*, *damaged*, and *dead*.

Steady: When the player's shields are fully charged, the shield meter will remain a consistent blue and be filled to the top.

Recharging: If a player is not at full health and their shield bar is recharging energy, the shield bar will smoothly fill up at the same rate that the player is recharging their shields. Also, when the player's shields are recharging, the shield meter will turn to a mild blue-green color to inform the user that their shields are recharging.

Damaged: When a player is damaged, their shield meter will snap to its new, damaged value. It should flash white in the same manner and timing as the tank icon does.

Dead: When a player is dead, the shield meter should pulse from deep red to red at the same rate as the player's tank icon. When a player is initially dead, for the first few moments, the shield meter will remain empty. As the camera moves towards the player's new spawn location, the shield meter should begin to fill with energy and reflect the amount of time left until the player spawns again. The meter should reach full at the same moment the player spawns, and immediately this meter should return to its steady, full color.

6.8.1.3 Energy Meter

A player's energy meter is an indication of how much energy they have available in order to fire a shot. The energy meter operates similar to the shield meter in that it changes color based on its status. If the energy meter is recharging, it will change to a mild blue-green color. The energy meter will also have subtle hash marks along it indicating where the energy will drop to if the user fires a shot.

6.8.2 Map

Within a mission, a player will have a map available to them to associate themselves with their position within the larger mission.

Bringing up the map will quickly present a 2D representation of the entire environment to the user. This map will show the locations of all friendly units and objectives. Objectives themselves will be listed under the map as well, with check boxes next to each one representing their completion.

Enemy units will not be visible on the map unless they are “in sight” of another friendly unit. Friendly units that are within range of enemy units will reveal the positions of the enemy units on the map. Once these enemy units travel out of site of the friendly unit, their position is no longer visible. When viewing the map, players should expect to see a solid front of friendly units, with enemy units popping in and out of visibility along the crest of this wave.

While viewing a map, the player is able to drive their vehicle, but unable to fire his weapon. While driving is possible, using the map while driving is not optimal, and is really only in place to allow the player to continue moving while viewing the map.

While viewing the map, icons on the screen will continue to update their position over time. They will not, however, reflect their direction. The map is only responsible for reflecting the position of objects in the world.

6.8.3 The In-Game Menu

A player is able to bring up a menu while in the game. In a single player situation, this action pauses the game (including audio) and the user will be unable to control their tank or affect the world at all until they un-pause the game. In a multiplayer game, bringing up the menu will still remove the player’s ability to interact with the game world, but the game world will continue as usual.

When the game is paused, the HUD and the 3D environment will fade behind the menu, bringing focus to the menu.

Options for the user within the menu include:

- **Resume Game:** Instantly hides the menu and returns the player to the game.
- **Exit to Menu:** Stops all sounds in the game and fades the screen out to black over a brief period of time.

6.9 Mission Statistics

After a player has completed or failed a mission, they are brought to the *Mission Statistics* screen. The mission statistics page will fade in after the game has faded out, and then quickly, item at a time, present the player with statistics regarding their performance in the mission. A player is also able to skip the introduction of statistics, quickly revealing all of the statistics to the player.

In a single player game, the bottom of the mission statistics state should provide the user the option to return to replay the mission if they have failed or return to the main menu. Should the player have completed the mission, the player will only be able to return to the main menu. This is to limit the possibility of a player accidentally repeating a mission that they have already completed, which is less likely to occur than if the player had failed the mission. If a user opts to replay the mission, the mission statistics screen will immediately disappear and they will be brought directly to the loading screen. If a player chooses to exit to the main menu, the screen will quickly fade to black and then enter the main menu state.

In a multiplayer game, a player is only able to return to the main menu from the mission statistics page, whether they completed the mission or not.

The statistics collected on a player's performance should be relevant to both the mission and to their individual contribution to the game. *Overall Mission Statistics*, or statistics that are not unique to each player, should be shown at the top of the mission statistics section, and *Individual Mission Statistics*, or statistics on the performance of each player in the game, should be itemized in a table for each player in the game.

6.9.1 Overall Mission Statistics

Overall Mission Statistics do not apply per player, but are instead the sum of all interactions that happened within the world.

- **Time spent within a mission:** The amount of time that a player spends in a mission. In a single player experience, this will not include time when the game is paused.
- **Number of Friendly Units Lost:** The number of friendly units killed by enemy units within the scope of the mission.
- **Total Enemy Units Destroyed:** The number of enemies destroyed by any member of the friendly team, including the NPC units.
- **Total Objects Destroyed:** The number of trees, buildings, fences, etc. destroyed.

6.9.1.1 Individual Mission Statistics

Individual mission statistics describe the performance of each player individually.

- **Number of enemies destroyed within a mission:** In single player mode, this is described by the number of enemies destroyed by the player, both as a sum of enemies destroyed and also broken down by weapon (primary gun and secondary gun).
- **Primary weapon shots fired:** How many shots were fired with a user's primary weapon.

- **Accuracy:** How accurate a user was. This is a combination of shots fired over units hit with a player's primary weapon.
- **Deaths:** How many times a player died within a mission.
- **Damage done:** How much damage was done, based on health points. This takes into account damage dealt by the player's primary weapon and secondary weapon.
- **Damage taken:** How much damage a player took over the course of a mission. This is based on health units.
- **Friendly fire:** How much damage a player did to friendly units
- **Friendlylies killed:** How many friendly units killed by a player.

6.10 Credits

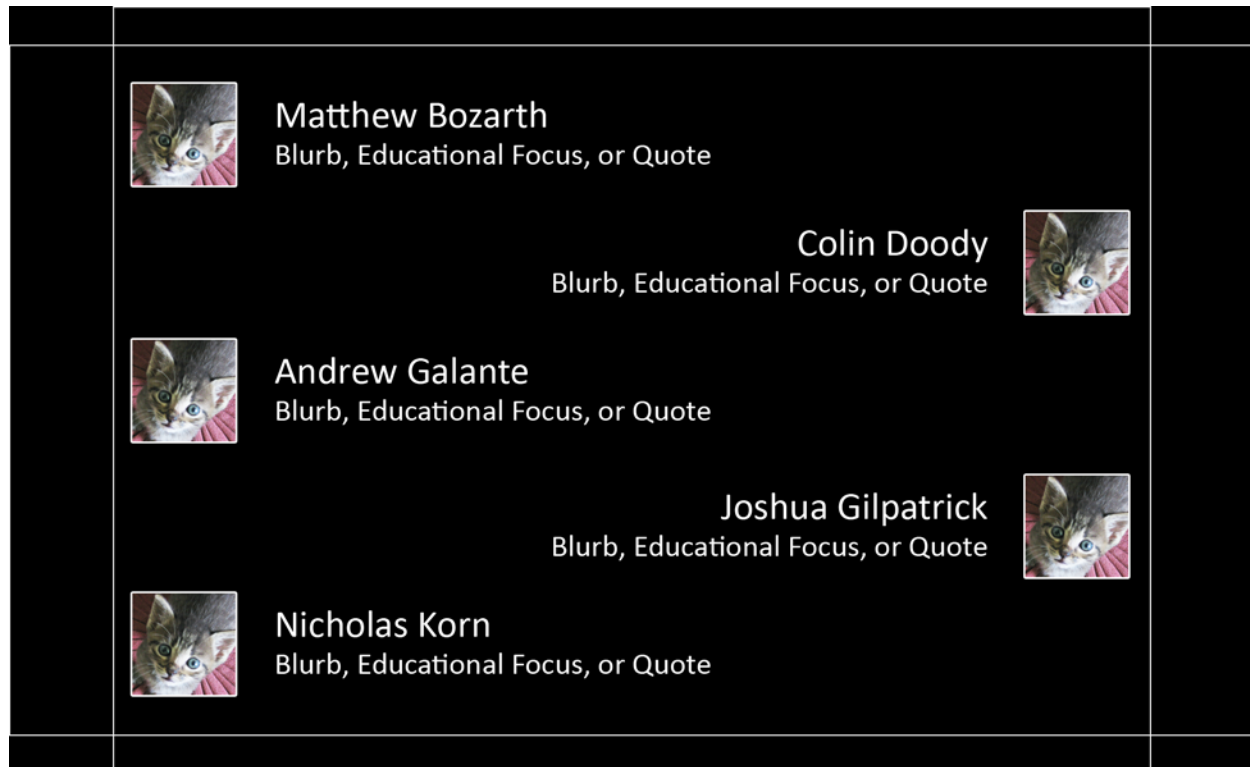


Figure 7.10-1: A mock up of the layout of the first stage of the credits screen

The credits state comes in directly after the player exits the game or when it is selected from the menu. Developers of the application and supporting faculty and programs are displayed in two stages. The two stages of credits are as follows:

- **Developers:** The students that made the game, listed in alphabetical order. Each student should have a picture, their name, and their primary responsibilities or research.
- **Assisting Faculty and Applications:** The following state should credit the faculty members directly involved with the program, the RIT Games Program itself, and should at the bottom list the icons of software that was involved in the development of the program.
- **Team and Contact:** The final stage of the credits should show the team that developed the game and contain contact information and a link to the website for the game design and development class of 2009.

The credits state should be skipped through quickly with user input, allowing players to exit the game in a timely manner. If the user selects the credits screen from the menu, the credits screen will return directly to the menu when it is done. If it plays when the game is exited, the game will exit once the credits are done.

7 Camera (Player View)

The camera is responsible for presenting the world to a player within a mission. The camera will view the player from a bird's eye view, looking straight down on the player. While playing a mission, the camera is tethered to the player and will always be moving with them.

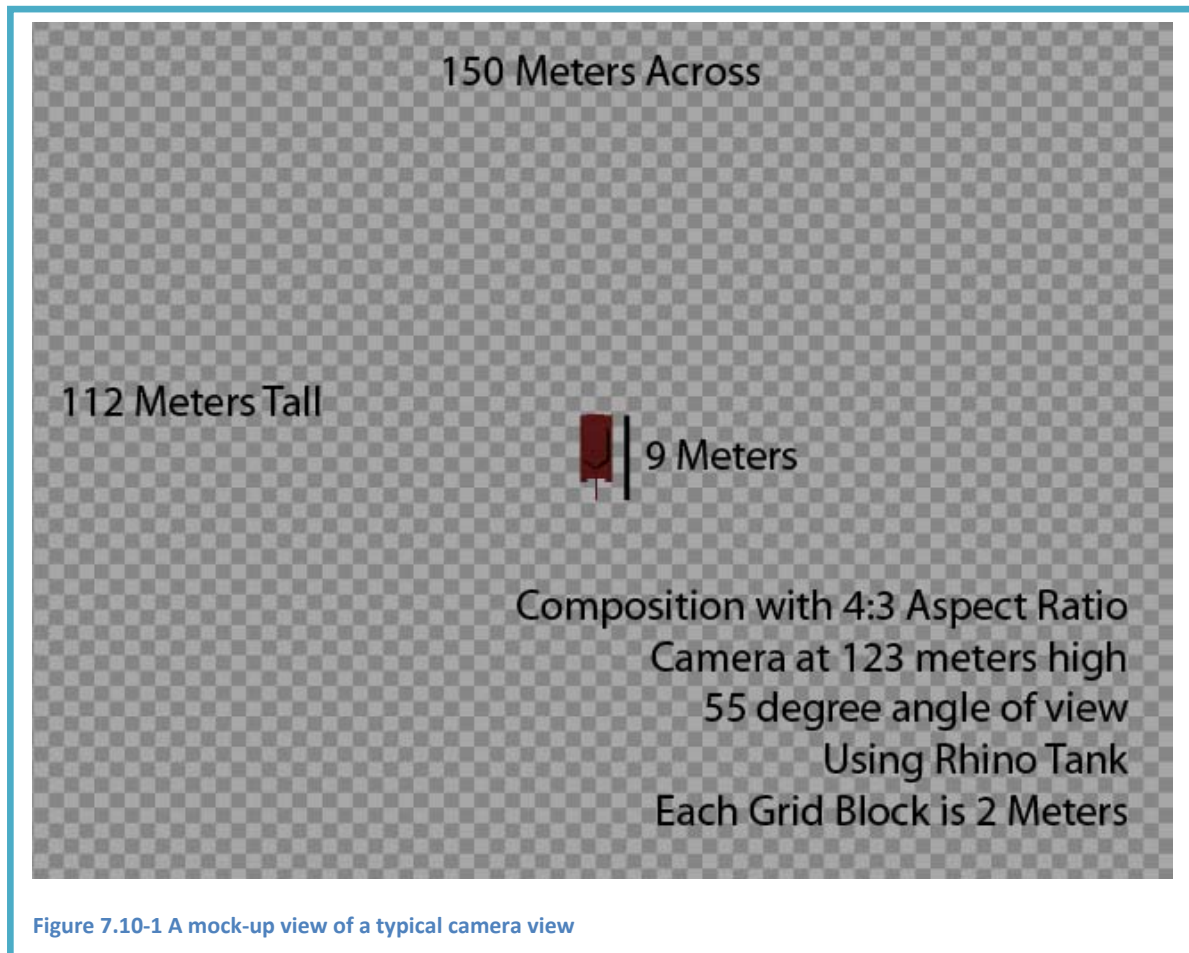


Figure 7.10-1 A mock-up view of a typical camera view

The camera needs to reveal enough about the player for them to understand their position in the world but also see enough of the surrounding terrain to allow a player to plan their moves and anticipate obstacles in the world around them.

For all player classes, the camera should by default reveal approximately 150 meters of surrounding terrain.

When a player plays as the Lancer tank class, their camera behaves slightly differently than other player's cameras. The player's camera will move from the same height as other players' camera to a height that reveals about one and a half times the terrain of normal players. When the Lancer moves, the camera attempts to be at the same height as other players' camera. When the Lancer sits still, the camera slowly moves back, revealing more terrain, until it reaches its highest point. If the camera is at

its highest point and the Lancer begins to move again, the camera slowly returns to its lower position. The movement of the camera up and down is a gradual, slow movement.

7.1 Mission Introductions

During the introduction of a mission, the camera will pan over key points in the map briefly, fading in from black and then back out to black. During this time, the camera is not locked above the player as it is in other situations. It moves smoothly and cinematically above several key points, looking directly down on the environment.

7.2 Death Sequence

When a player dies, the camera is responsible for giving the player a moment of time to understand where they died and what happened when they died, and then pan over to their new spawn location. The camera must move gradually to the new spawn location so the player has an understanding of where their new location is relative to where they died in the world. The camera will also pull back slowly at the start of the death sequence, giving the player a floating sense of death, and zoom back into the world as the player is respawned.

8 Audio

Audio plays a role in the game in not only setting mood and ambience, but also alerting players to events that are happening in the world and giving them non-visual cues regarding their condition or progress in the mission.

Audio can be found throughout the game in the form of sound effects, music, and dialogue from the commander.

8.1 Sound Effects

There are many types of sound effects that are part of the game. Certain sound effects will only be played once, but others will loop indefinitely or until a certain situation is reached. Sound effects can be found throughout the player's interaction with the user interface, in the game world, and through game notifications.

8.1.1 User Interface Sound Effects

User interface sound effects are played in response to actions on the user interface. These sound effects exist in the menu states and, to a limited extent, in the game. In the game, they are be used in coordination with the game menu. They are generally quick and reflect the type of animation that is occurring with the interface itself. For example, if a menu item flashes white quickly when a mouse hovers over it, the sound that accompanies the flash needs to evoke similar quick, bright sounds.

Required User Interface Sound Effects

Menu Item Highlighted <i>Subtle, Soft, High Pitched</i>	Used when a menu item is highlighted. This can be due to a mouse hovering over the item or the keyboard being used to highlight an item.
Menu Item Clicked <i>Bold, Quick, Forward</i>	When a menu item is selected, a sound effect should be played indicating that it has been selected.
Going "Backwards" <i>Backward, Fading, Depressed</i>	Used specifically in menu situations where a user has gone "backwards" from their current position. This is the case when a submenu is closed from within a single screen or a dialogue box is closed.
Typing <i>Quick, High-Tech, Sharp, Quiet</i>	A simple sound effect should be used when typing occurs to portray a futuristic feel.
Connecting	A looping sound effect to be played during a connection dialogue.

8.1.2 Action / World Sound Effects

Action / World sound effects are effects that exist within a mission that are activated by an event or object within the world. These sound effects are only generated by units (NPC and player units) or events within the world.

Sound effects that occur within the world exist in the three-dimensional space that occupy. A player will only hear sounds that are triggered near them, and the pitch, volume, and frequency are adjusted according to the Doppler Effect.

Required Action / World Sound Effects

Vehicle Driving	Each vehicle will use a different pitch. The heavier the vehicle, the deeper the pitch of the driving noise. Driving sounds should resemble modern world tanks or large machinery.
Cannon Firing	Each class will have a different pitch of cannon firing. Should be a deep, powerful sound. Less of an explosion noise and more of the noise of a shell exiting a cannon.
Machine Gun Firing	A smooth, constant sound of machine gun fire. A high pitched whir coupled with a buzzing of gun fire.
Machine Gun Hitting Metal	A light, pinging noise. Should not trigger as frequently as the surface is being hit, but only once in a while to avoid too many sounds being played at once. Should also vary in pitch to avoid redundancy.
Machine Gun Hitting Snow	A softer smacking noise. Similar to the gun hitting metal sound, it should not play too often and should modulate in pitch.
Shell Hitting Metal	A hard, sharp, fast explosion sound. Should not reflect the sound of metal at all, but should sound more like an explosion would on any surface.
Shell Hitting Ground	A soft explosion sound. Similar to the ground

	noise, but with a more muffled feel to it.
Explosion (of the player)	A long explosion. Should be similar to the explosion of another unit, but should last longer and be more involved, to let the player know that they are the ones that have been destroyed in this case.
Explosion (of another unit)	A hard, medium length explosion sound. Should have small qualities of bending metal and also the generic noise of an explosion.

8.1.3 Environmental Sound Effects

Environmental sound effects exist within the world and support the effort in creating an immersive environment. For example, when a player is near a river, they should hear the sound of the river trickling by. When a player is near woods, they should hear the sound of trees rustling and wood creaking. When near an industrial area, they should hear the soft ambience of machines working and steam escaping pipes. Environmental sound effects are subtle and aim not to overwhelm the music or sound effects in the game. They become quieter as a player leaves an area where an environmental sound exists.

8.1.4 Notification Sound Effects

Notification sound effects are effects that come up in the game that do not exist in the world but notify the player of game specific events occurring around them. Notification sound effects should typically be brief, but sharp enough to notify the player of an event in the world.

The following lists the types of situations that would call for a particular notification sound effect:

Required Notification Sound Effects

Objective Completed	Plays when an objective has been accomplished by anyone on the friendly team. A high pitched chiming indication of the completion of a task.
Game Paused	When a player pauses the game in single player mode. A deep thumping sound to accompany the abrupt freezing of the game.
Game Resumed	When a player resumes the game from single player mode. A sort of inverse thumping sound, to portray the effect of the world resuming around the player.
Death	When the player dies. A fading, deactivating sound alluding to the lack of accomplishment.

Respawn	When a player is respawned. A growing, reactivation noise coupled with the sounds of machinery coming online.
(Multiplayer) Teammate Lost	When a teammate is lost in a multiplayer game. A shorter version of the player death sound, more abrupt, but invoking the same sense of loss.
(Multiplayer) Teammate Respawn	When a teammate has respawned. A shorter version of the respawn sound, alerting the player to the rebirth of another teammate.

8.2 Dialogue

Dialogue is speech from a faceless commander that gives the player feedback regarding their condition in a mission. The commander is a female that is never introduced to the player formally in the game, but delivers messages to the player throughout multiple stages of a mission. The dialogue is presented to the user with a calm but assertive voice, coupled with high frequency radio static, to give the player the impression that commands from the commander are being delivered remotely through a radio control. The commander is responsible for giving the player their objectives at the beginning of a mission and also alerting the player to enemy activity or events in the game.

Some dialogue that the commander delivers is generic, and can be applied to any mission, and other dialogue clips are specific to the mission itself.

8.2.1 General Dialogue

General dialogue is common throughout all missions. The following general dialogue clips can be found throughout the game:

Required General Dialogue	
When a mission is won	"Great job. Now pack it up and come home. Tomorrow is another day ..."
When a mission is lost (through death)	<i>[with static]</i> "Unit, we're losing your signal. Report. Report!"
When a mission is lost (by losing an objective)	"Our objective has been compromised. We've failed the mission. Minimize losses and bring it home."
When an objective is completed	"Objective complete. Good job!"

8.2.2 Mission Specific Dialogue

Each mission can define its own dialogue that can occur during an event or during the introduction sequence of a mission. Each mission has a sequence of dialogue to accompany the introduction sequence of objectives. Event dialogue is also specific to the mission and can alert the player to new events occurring or unfolding in the environment.

For example, if a player enters a region that introduces a number of enemy units, the commander may tell the player *“I’m picking up on a number of enemy units heading your way. Stay alert!”*

8.3 Music

The game uses music to help set its mood and atmosphere. Music plays throughout the game, from the moment the game is started to the end of the credits sequence. Music is ambient throughout the game and aims to not distract the player.

The game’s menu (all phases except for the game phase and the mission statistics phase) uses one song. This song is less ambient than the others and aims to create more of a memorable melody than other music that is found in the game. This music will only restart when a player returns to a menu phase from a game phase.

Each level in the game has its own song to reflect its specific setting and atmosphere. A level with an industrial setting, for example, would be filled with more heavy percussive elements than a level that takes place in a desolate, icy gulch. The music for each mission will play from the start of the mission until the player exits the game or is victorious or defeated in a mission.

When a player fails a mission or completes a mission, the victory or defeat music plays after the end-game dialogue. This music will interrupt any music that was playing during the game, and can be more melodic and catchy than other sounds. These tracks should aim to reflect the mood of victory or defeat, either through powerful major sequences or slow, minor tones.

Thematically, the music will mix industrial metal sounds, massive orchestral instruments, and a touch of electronic leads to help portray the concept of a futuristic battle in a war torn world.

The following list describes the various musical tracks needed for the game, and the theme they aim to achieve:

Required Music

Menu Music	Heavy, chugging guitar. Massive and orchestral. Fast. Exciting. Epic. Heavy on percussion as well.
Level Music	Specific to the level. Fast paced, but tenser than the menu music. Still metal and orchestral. Less epic. Dependent on the theme of the

	environment.
Victory Music	More orchestral and less metal. Epic. Shorter.
Defeat Music	Slower, slightly depressing and soft. Orchestral. Heavy on bass sounds and percussion.

9 Multiplayer Experience

Players engage in a multiplayer game in one of two ways. The first is to setup a local area network (LAN) connection that will be optimized to send data to players on a local network. The other way is to set up a game using TCP/IP. The latter way will require knowledge of the users internet protocol (IP) address. Up to four players can play the game cooperatively at a time in a multiplayer game.

When setting up a multiplayer game, players are able to assign themselves an alias, which is their identity and how others see them in multiplayer games. They are also able to communicate with one another through text messages when setting up a game and while playing the game.

9.1 Alias

Before joining a multiplayer game, a user will be required to enter an alias. This alias serves as a way to identify players in a multiplayer situation. The alias will be required before the player joins the “Team Game Selection Room” from the main menu.

9.1.1 Conflicts in Alias Names

Although a user is able to enter any alias when they enter the *Team Game Selection* room, a conflict may occur if two users have entered the same alias and join the same *Team Game Setup* room. Should this occur, the second player to enter the Team Game Selection room should have an identifier appended to their name. Upon exiting the room, the player’s name should return to the alias they had originally defined.

9.1.2 Predefined Alias Names

The following is a list of predefined alias names that are provided automatically and at random when joining the *Team Game Selection* room from the menu. The player will be able to replace these with their own if they desire. The predefined alias lists lets a user quickly jump into team play without needing to spend time entering their name.

- Shadow
- Renegade
- Messiah
- Checkmate
- Cloverleaf
- Duke of Terror
- Eagle Eye
- Ginger
- Road Hog
- Rapture
- Beagle-Face

9.2 Communication

Players have the ability to communicate with one another in the Team Game Setup room and within a mission through text messages. A text message will be sent to all players that are connected to the game. A message will start with the name of the player who sent the message and then follow with the message itself. A player who sends a message will also see the message that they send with their name in front of it, but it is colored uniquely from other user messages to let the user know that it is his own.

9.3 Leaving the Game

If a player leaves a multiplayer game, whether it is intentionally or unintentionally, it must be presented to the other users in the game in an obvious way to let them adjust accordingly.

9.3.1 Team Leader Disconnection

When the team leader disconnects from a game, the game will no longer continue for players due to their dependence on the team leader.

Within a Mission

Should the leader of a team game exit the game in the middle of a mission, the game will end for all users connected to the game. The sequence should be graphically presented in the same way that a player death occurs (the screen should turn red and the player loses the ability to control his tank) and a special menu should appear describing that the team leader has left the game. There is one option for the user to select at this point, allowing for them to return to the “Team Game Selection” room.

Important Dialogue for Team Leader Disconnections within a Mission

Message	“The team leader has left the mission”
Option	Leave the Mission

Within the Mission Info / Game Loading screen

If the team leader disconnects within the *Mission Info / Game Loading* screen, the screen will fade to black over a short period and return the user to the *Team Game Selection* room. After the quick introduction to the *Team Game Selection* room, a dialogue box will appear instructing the user that the team leader has exited the game. The dialogue box should contain a button allowing the user to continue.

Important Dialogue for Team Leader Disconnections from the *Mission Info / Game Loading* screen

Message	“The team leader has left the mission”
Option	Continue

Within the Team Game Setup room

If the team leader disconnects from the *Team Game Setup* room, the *Team Game Setup* room should transition out as if the user had exiting the room normally and return to the *Team Game Selection* room. In the same manner as the Mission Info / Game Loading screen, the Team Game Selection room should transition in and a dialogue box should appear informing the user that the team leader is no longer part of the game.

Important Dialogue for Team Leader Disconnections from the *Team Game Setup* screen

Message	"The team leader has left the room"
Option	Continue

9.3.2 Teammate Disconnection

When a teammate leaves the game that is not the team leader, the game can continue as normal for other players with a few important notifications for the other players.

Within a Mission

If a player leaves within a mission, the game can continue as normal for other players. A message should come up in the chat window informing players that the user has quit, highlighted in a color that is different than the other chat colors. The user's tank should disappear from the world and his team icon should be presented in a way that is similar to that of him being dead without the ability to respawn.

Within the Mission Info / Game Loading screen

If a player leaves within the mission info / game loading screen, the game should continue as normal without any indication of his absence until the mission starts. When the mission starts, the same notification and actions should occur as though the user had exited from within a mission.

Within the Team Game Setup room

If a user disconnects from the *Team Game Setup* room, a message should appear in the chat window in the *Team Game Setup* room indicating that they have left. His player icon should also be removed from the Team Game Setup room and they are no longer required to be ready in order to start the game.

10 Conclusion

With a refreshing coop campaign, unique classes encouraging a variety of play styles, and open ended missions, we feel this game is going to bring a style of gameplay sorely missed from the current marketplace and are eager to begin development.

11 References

- [1] The DeGraeve Color Palette Generator
<http://www.degraeve.com/color-palette/index.php>

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Level Design Document

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Level I: Fort Ashford

Front line battle in enemy territory

Fact List

Mission Overview

Mission Title	Fort Ashford
Mission Brief Description	Front line battle in enemy territory. Sever communications and destroy new energy reserves. Gather recon on their infrastructure. Bring the battle to their homes.

Objectives Overview

Destroy All Oil Tanks	Destroy all 10 oil tanks located around the map
Upload Information from Within the Barracks	Stay alive within the enemy barracks in the north for 30 seconds
Destroy the Communications Tower	Destroy the single heavily guarded communications tower on the peninsula in the north east, as well as related construction units

Events Overview

Enter the Barracks	Once the barracks are entered, enemy units come in from Enemy Spawn II to stop you.
Attack the Communications Tower	Once the communications tower is struck, enemy units spawn from the north and south at Enemy Spawn III and Enemy Spawn IV, heading towards the peninsula.

Overview

It is time to deal a crippling blow to the enemy. We are sending a battalion to their northern Fort Ashford. We need destroy several of their key reserves in the area. We also need to collect recon on some of their internal structures in the north. The center of the region is populated with military structures and defenses. The outskirts are rugged, but our surveillance has pointed out that some of their reserves are being stored in these areas.

Map



Objectives

Destroy the Oil Refineries

The friendly team must destroy all 7 oil tanks located around the map. They are located on the map as red triangles. Friendly units are able to complete this task.

Gather Information from within the Barracks

The player must enter the enemy barracks and stay alive within the walls of the barracks for thirty seconds. The player must reach the center of the walled area for the objective to begin. Once the objective has begun, the player must remain within these barracks for thirty seconds. If a player is not present within the walls of the barracks, the objective is failed. When the objective begins, an event is triggered that sends two waves of enemy units towards the barracks.

Destroy the Communications Tower

On the north east peninsula there is a new communications tower being built. Destroy the communications tower and all surrounding construction units. The communications tower is heavily guarded with turrets and other units. The surrounding construction units should be quick to destroy. The communications tower is not only strong, but well guarded. Attacking the communications tower will trigger a wave of enemy units that will come from both ends of the peninsula. The communications tower is a magenta icon.

Events

Enter the Barracks

Upon entering the barracks, an objective will be triggered that requires the player to remain stationary inside the walls of the barracks for thirty seconds. This event will also trigger two waves of enemy units.

Attack the Communications Tower

Once the communications tower is hit, enemy units spawn in the north and south and collide on the position in the peninsula in an attempt to stop your destruction.

Audio

Music

The music in this level should have an industrial sound. It should be less eerie than an open environment would be. This can be attributed to the region being populated and presumably an enemy base.

Dialogue

Introduction Dialogue: "Welcome to combat soldier. Here's your brief: There are several oil refineries located in this region supplying the enemies factories. Now's our chance to cut off their supplies. Take out all seven oil refineries that we've been able to identify. There's a communications tower going up in the north east that could cause us some trouble later, so do your worst over there. Also, intelligence wants some recon from inside the enemy barracks. See what you can get for us."

Gather Information Event Dialogue: "I was afraid of this. They've got reinforcements heading your way. Intel wants you to stay put while we capture your upload. Try to hold them off. Don't leave those barracks."

Attacking the tower: “I’m picking up activity from your north and south. Be careful.”

Environmental Audio

Stream water: The sound of a stream should be placed along the river in the north.

Industrial sounds: Industrial sounds can be found in the south east and on the peninsula near where the tower is being erected.

Wooded area noises: To the west is a wooded area. Throughout this area should be the sounds of trees rustling in the wind.

Oil refinery noises: In the middle of the map should be the sounds of an oil refinery. An oil refinery has a more massive machinery sound than the regular industrial zones.

Curiosity Studios

Working Title: Tank Frenzy

Technical Design Document

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

1	OVERVIEW	100
2	HIGH LEVEL CONCEPTS.....	100
2.1	CODING GUIDELINES.....	100
2.1.1	<i>General Guidelines</i>	100
2.1.2	<i>Documentation</i>	101
2.1.3	<i>Style</i>	101
2.2	DIRECTORY STRUCTURE	102
2.3	FILE NAMING SCHEME	103
2.3.1	<i>Code Files</i>	103
2.3.2	<i>Asset Files</i>	103
2.4	COMPILATION	104
3	MARABUNTA ENGINE	105
3.1	MARABUNTA CORE	105
3.2	SOUND SYSTEM	106
3.2.1	<i>SoundSystem</i>	106
3.2.2	<i>ISound Class</i>	107
3.2.3	<i>Sound2D Class</i>	107
3.2.4	<i>Sound3D Class</i>	107
3.3	GUI SYSTEM	107
3.4	NETWORKING SYSTEM	108
3.4.1	<i>Client</i>	108
3.4.2	<i>Server</i>	108
3.4.3	<i>Packet</i>	109
3.5	SCRIPTING SYSTEM	109
3.6	INPUT SYSTEM.....	110
3.7	WINDOWING SYSTEM	110
3.8	RENDER SYSTEM	111
3.8.1	<i>Scene Management</i>	111
3.8.2	<i>Scene Graph</i>	115
3.8.3	<i>Spatial Graph</i>	117
3.8.4	<i>Render Graph</i>	119
3.8.5	<i>Render Targets</i>	119
3.8.6	<i>Bounding Box</i>	120
3.8.7	<i>Particle System</i>	120
3.8.8	<i>Other Effects</i>	125
3.9	RESOURCE SYSTEM	125
3.9.2	<i>Shader Manager</i>	128
3.9.3	<i>Audio Files</i>	128
3.10	CONSOLE SYSTEM	128
3.11	MATERIAL SYSTEM	129
3.12	DEBUGGING SYSTEM.....	129

3.13	TIMER SYSTEM	130
4	GAME TECHNOLOGY	131
4.1	MAIN	131
4.2	TANK MANAGERS	131
4.3	131
4.4	STATE SYSTEM	131
4.5	ISTATE	132
4.5.1	Logo State	132
4.5.2	Menu State	133
4.5.3	Credits State	133
4.5.4	Options State	134
4.5.5	Team Game Selection State	134
4.5.6	Team Game Setup State	135
4.5.7	Solo Game Setup State	135
4.5.8	Load State	136
4.5.9	Play State	137
4.6	INFLUENCE MAP	137
4.7	AI SYSTEM	137
4.7.1	High Level Decision Making	137
4.7.2	Low-Level Behaviors	138
4.8	THREADING FOR AI	141
4.8.1	Thread	141
4.8.2	Thread Arguments	141
4.8.3	Thread Pool	141
4.8.4	Task Manager	141
4.8.5	Task Interface	142
4.8.6	Shared Memory Interface	142
4.8.7	Double Buffer Interface	142
4.9	UNIT	143
4.9.1	UnitManager	143
4.9.2	Attributes	143
4.9.3	Functions	144
4.10	WEAPON	144
4.11	BULLETS	145
4.11.1	BulletManager	145
4.12	CONTROLLER	145
4.12.1	Player	145
4.12.2	NPC Controller	145
4.12.3	Turret Controller	146
4.13	DESTRUCTIBLE OBJECT	146
4.14	INDESTRUCTIBLE OBJECT	146
4.15	OBJECTIVE	146
4.16	MISSION	146
4.16.1	Mission File Syntax	146

4.17	LEVEL.....	147
4.17.1	<i>Level Parsing</i>	147
4.17.2	<i>Level File Requirements</i>	147
4.17.3	<i>Level File Syntax</i>	147
4.18	SESSION	148
5	NETWORKING PROTOCOL	148
6	CONTENT	149
6.1	INFLUENCE MAP PARAMETERS	149
6.2	COMMANDER AI PARAMETERS (HIGH LEVEL)	149
6.3	LOW LEVEL AI PARAMETERS.....	149
6.4	CONSOLE COMMANDS	149
6.5	USER CONFIGURABLE GAME OPTIONS.....	150
6.6	USER INTERFACE COMPONENTS	150
6.7	CONTROLS	150
6.7.1	<i>Tank Controls</i>	150
6.8	NPC CLASSES.....	151
6.8.1	<i>Wolfhound</i>	152
6.8.2	<i>Durant</i>	153
6.8.3	<i>BRAMM</i>	154
6.8.4	<i>Stonewall</i>	155
6.9	PLAYER CLASSES.....	156
6.9.1	<i>The Hornet</i>	157
6.9.2	<i>The Rhino</i>	158
6.9.3	<i>The Lancer</i>	160
6.9.4	<i>The Ant</i>	162
7	EXTERNAL TOOLS / UTILITIES	163
7.1	LEVEL EDITOR.....	163
7.2	MAYA EXPORTER	163
7.3	FONT SHEET GENERATOR.....	163

1 Overview

This document is the official technical design document for our game. It will not contain elements specific to our capstone game play. There are four sections in this document,

- High Level Concepts – This section talks about the coding standards and guidelines the developers are going to follow when developing the game and engine. These standards are set in place to make code easier to read and if people need to shift projects they will be familiar with the layout of any new code they transfer to.
- Marabunta – This section talks about the implementation details for the 3D rendering engine for our game. The details will talk about the specifics behind the rendering pipeline.
- Game Engine – This section will talk about the components that are needed for creating our game. Each component will describe how it works and how it interacts with the Marabunta engine.
- Tools – This section talks about the external tools that have been built to assist development. These tools describe any protocols needed to interface with the tool.

2 High Level Concepts

This section consists of coding and naming guidelines that will be followed during the development process of our game. While there is no direct punishment for not following these guidelines and we have no easy way to enforce them, a good talking with the person who broke them and forever making fun of them will be their punishment.

2.1 Coding Guidelines

Coding guidelines are established to create a uniform working environment that all programmers can understand. This will help ease the understanding of others code, while making sure that when we send our code to companies for review it is clean and understandable.

2.1.1 General Guidelines

Objects should be deleted in the scope that they are created. This maintains a clear ownership of objects at all times. An analogous policy will also apply to the release of COM objects. If an object needs to be passed between other objects, use references, double pointers or accessor functions to maintain the object ownership rules.

If a class is to be sub-classed, it must have a virtual destructor. Singleton classes should have private constructors and destructors to prevent accidental instantiation or inheritance.

2.1.2 Documentation

Documentation is important in this project, particularly within the engine itself. Documentation will be maintained using *Doxygen*, so all classes, public methods and members should be documented in a format recognized by Doxygen. Things that will be documented this way are classes, and functions. Classes will have a standardized header that is detailed below

See <http://www.stack.nl/~dimitri/doxygen/docblocks.html> for more information.

The following is an example of commenting a function using these guidelines.

```
/**
 * A short description in the first sentence.
 * Followed by a more detailed description, in one or more sentences.
 * @param p1 Function parameter 1.
 * @param p2 Function parameter 2.
 * @return The function return value.
 * @see a_related_identifier
 */
int function(int p1, int p2);
```

Class header comments

```
/**
 * Creator: Your Name
 * Description: A description of what the class does
 * Revisions: 1 - change this every time the class is changed and
checked in. don't bother with sub numbers just 1,2,3 this is so we can
track it in the svn a bit easier
 * Log:      1:2/11/09 - created
 *           2:2/12/09 - fixed bug
 */
```

2.1.3 Style

This section follows the format of, and borrows heavily from, the C++ coding standard found at: <http://possibility.com/Cpp/CppCodingStandard.html>. Coding styles are fairly diverse, as are their reasons. However, it is important to maintain a consistent coding style across the project to enhance readability and maintainability. This is especially true of header files, which serve as interfaces to the classes they declare.

The coding style listed here is only a guideline, and departures from it will be allowed within reason.

2.1.3.1 Naming Conventions

Enumerations, Constants, and #defines: These objects should be written in all capitals, to emphasize that the value is immutable. Spaces between words should be separated by underscores.

Example: `#define NUM_ELEMENTS 2`

Classes and other type names: These types should be spelled out in Pascal-case.

Example: `Tank, FooBall`

Interfaces: Interfaces should be spelled out in the same manner as classes, but should be prefixed with a capital “I”. This makes semantic meaning of an interface more apparent.

Example: `IEntity, IRenderableNode`

Class Methods and Functions: These should be Pascal-cased in the same manner as classes. This is to differentiate between members and functions.

Example: `Render(), GetFoo()`

Class Members: Should be all camel-cased prefixed with an underscore (`_`) to denote that it is a class variable opposed to a local variable. Also variables should be assigned with like variables and tab aligned to the shortest length.

Example:

```
int         _health;
double      _ammo;
double      _spawnRate;
D3DXMATRIX _world;
```

Global Variables: These variables should be avoided at all costs; in the event one is required it should be denoted with “`g_`” to denote that it is a global variable.

Example: `g_device`

2.2 Directory Structure

Since this game will be comprised of many components, it would be prudent to organize them into a directory structure. Inside the main source tree, at least these four directories should exist: Game, Source, Engine, and Documentation. These aren't required names; they are used here to illustrate their function. “Game” contains compiled versions of all game assets, including executables, textures, shaders, and parameter files. Once built, this folder should contain everything needed to run the game. “Source” contains game asset sources, including but not limited to source code, Photoshop .PSD files, FBX models and other high-detail art assets. During the build process, Visual Studio will be configured to place game binaries in the “Game” folder, while other game assets should be placed manually. “Engine” contains engine source code that isn't specific to the game being created. This code will be linked with the game binaries during the build process. “Documentation” contains Doxygen-generated html files for the engine and game code. There will be a script supplied in the project folder to run Doxygen and create this documentation.

Inside the “Game” folder, and “Source” where applicable, the following folders shall be created:

GUI	Game GUI assets
Models	3D models and meshes, both animated and static
Music	Background music for the game
Scripts	LUA script files
Shaders	HLSL FX files used by the render system
Sounds	Sound effect clips
Textures	2D assets (skins, heightmaps) not part of the 2D GUI system
XML	XML configuration files

2.3 File Naming Scheme

2.3.1 Code Files

ClassName.cpp, ClassName.h

2.3.2 Asset Files

Asset files should follow a naming scheme as they enter the engine in order to ensure that they are easily accessed and easily loaded throughout the development and implementation of the application. As these items are included in the project they will be placed in appropriate folders.

2.3.2.1 Audio Files

Audio files can be one of two types of file: a sound effect and a music file. The game engine itself is unaware of the differences between the two. The difference in the naming of the two types of files is to help assist the developer implementing them.

Audio File Name Examples

Music

Music_TitleScreen.mp3

Sound Effects

Sfx_Action_TankFire.wav

Sfx_Env_River.wav

Sfx_UI_ButtonDown.wav

Dialogue

Dialogue_Level01_Intro.wav

Dialogue_Generic_MissionFail.wav

2.4 Compilation

The game and engine will be compiled for both 32 and 64 bit modes. Code should be written with this in mind, and therefore not make assumptions about type sizes. If only 32 bit development machines will be provided, personal machines that run in 64 bit mode may be used. Before any check-ins of source code, the code must compile and run without any detectable memory leaks. Visual Leak Detector will be used for this purpose. It should be configured to compile in 32 bit debug mode because VLD currently doesn't support 64 bit, and so that it doesn't stay active in release mode.

3 Marabunta Engine

Our engine will be in its own project outside of our game project. The engine will compile to a dynamic linking library (DLL) that will link up with the game project and the development of the systems that make up our game will need to take this into account.

The engine itself will be broken down into various components that act as a sub system to the engine. These components are listed below, and describe in detail further in the document. Each sub system listed will be compiled as part of the engine DLL with the exception of the GUI. This sub system will be compiled into its own DLL and be included in both the game and the engine.

Engine Subsystems

- MarabuntaCore
 - Windowing System
- Scene graph
- GUI
- AI
- Networking
- Audio
- Input
- Console (text/scripting)
- Lua State Management

3.1 Marabunta Core

The core system will need to be initialized before using any aspects of the engine. This initialization call will create a DirectX device, and initialize the subsystems of the engine. After this has been initialized you are able to use the full functionality of the core including setting up a window, and any subsystems that are included in the engine.

MarabuntaCore
<u>+Initialize() : MarabuntaCore</u>
+CreateWindow()
+CreateServer()
+CreateClient()
+GetWindow()
+GetDevice()
+GetInputSystem()
+GetSoundSystem()

3.2 Sound System

The audio system will be accessible through the core engine, support its own resource management of audio files, and have functions available for playing audio in 2D and 3D space. The audio system will have the ability to play multiple sounds and control various properties of sound, such as volume and streaming.

The sound system as a whole consists of four classes:

- **SoundSystem:** The core of the sound system. Only a single instance of this class will exist at a time. It will be initialized and updated through Marabunta, and is the factory for other sound files.
- **ISound:** The ISound class is an interface that maintains common functions between the two Sound class types.
- **Sound2D:** The Sound2D class is a sound class that exists in 2D space. It is not updated or controlled in the same way as the 3D sound class.
- **Sound3D:** The Sound3D class is a class that exists in 3D space in the world. Unlike a Sound2D class, it has position information.

3.2.1 SoundSystem

SoundSystem
<u>+Instance() : SoundSystem</u>
+Initialize()
+LoadSound()
+UnloadSound()
+PlaySound()
+PauseSoundState()
+StopSound()
+PauseAllSoundsState()
+StopAllSounds()
+Update()
+GetSound2D()
+GetSound3D()

The SoundSystem class, which is the core of the entire sound system, is the manager responsible for loading sound resources, creating sounds, and updating sounds.

The sound system will manage its own sound resources.

3.2.2 ISound Class

Sound
+SetVolume() +SetLoopState() +SetChannel() +PlaySound() +StopSound() +PauseSoundState()

3.2.3 Sound2D Class

Sound2D:Sound
+SetPan() +GetPan()

3.2.4 Sound3D Class

Sound3D:Sound
+SetPosition() +GetPosition()

3.3 GUI System

The GUI system exists in its own project

The GUI system is initialized by Marabunta at the creation of the engine. The scenes within the GUI system will be generated by the application itself. The GUI system is accessible through the Marabunta core.

GUI	Scene	Panel	AnimatedPanel	TextField
+CreateScene() +GetScene() +Update() +LoadLayoutTemplate() +LoadLayoutTemplateResources() +UnloadLayoutTemplate() +UnloadLayoutTemplateResources() +LoadTexSheetTemplate() +LoadTexSheetTemplateResources() +UnloadTexSheetTemplate() +UnloadTexSheetTemplateResources() +PushMouseDown() +PushKeyboardInput() +SetKeyboardFocus()	+SetSize() +GetSize() +SetSizeScaled() +SetRenderTargetType() +Set() +GetRenderTargetType() +AddChild() +GetChild() +RemoveChild() +DeleteChild() +ActivateTemplate() +DeactivateTemplate() +CreateTween() +CreateElement() +SetPaused() +InjectMouseDown() +SetMouseEnabled() +GetMouseEnabled()	+SetTexture() +SetX() +SetY() +SetPosition() +SetScaleX() +SetScaleY() +SetRotation() +Rotate() +SetColor() +AddChild() +RemoveElement() +DeleteElement() +Visible() +SetOrigin() +SetAlign()	+GotoAndStop() +GotoAndPlay() +SetLoop() +Stop() +Play() +Pause() +SetLoopCount()	+SetFont() +SetText() +SetHighlight() +SetWidth() +SetHeight() +SetLineSpacing() +SetLetterSpacing() +GetText()

3.4 Networking System

The server will use a combination of TCP and UDP connections with the possibility to create a multicast group.

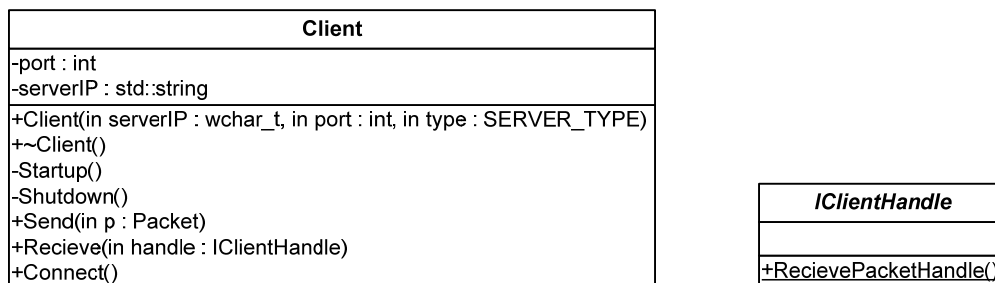
UDP will be used to send game state packets, to keep the clients in sync with the server. The advantage of using this protocol is because game state packets are time-critical, but not ordered. Packets should be processed quickly, and missing or dropped packets aren't critical enough to be resent.

The reason for opening a TCP connection is to transfer communication data, such as joins, leaves or chat. This data must be sent reliably between the server and clients.

OPTIONAL: Multicast can be implemented if enough development time is available. The benefit to using a multicast group means that fewer packets need to be sent across the network from the server. However play will most likely be restricted to a LAN because many routers do not forward these packets. This will be a configurable option when creating the server. When enabled, the server will send all game state packets to the multicast address. Clients must still send their update data to the server's unicast UDP port.

3.4.1 Client

The client will use the UDP or TCP protocols to talk with the server. Depending on the constructor called, the client will toggle between joining a multicast group, a UDP connection that relies on endpoints or a TCP connection. These objects exist only on the client executable instance.



3.4.2 Server

The server will be setup in a similar way the client. The constructor will determine what the server is setup for (TCP/ UDP / multicast). A game can have multiple servers running. The server will setup a keepalive thread that will send out a packet at a set interval. This can be toggled on or off. If this is toggled on then a keepalive packet will be sent to the client and the client is expected to respond within two packet sends.

Server
-port : int -bindIP : std::string -connectedClientList : std::map
+Server(in port : int, in type : SERVER_TYPE) : Server +Server(in bindAddress : std::string, in port : int, in type : SERVER_TYPE) : Server +~Server() -Startup() : bool -Shutdown() : bool +Accept(in handle : IServerHandle) : void +Send(in id : int, in p : Packet) : bool +SendAll(in p : Packet) : bool +Recieve(in id : int, in handle : IServerHandle) +Recieve(in handle : IServerHandle)

IServerHandle
+NewClientHandle() +RecievePacketHandle()

3.4.3 Packet

This section will specify the packet layout.

Packet
-type : int -length : int -data
+SetData(in type : int, in length : int, in data) +GetData() : void +GetType() : int +GetLength() : int

3.5 Scripting System

A scripting system will be used to maintain and interface with the Lua state.

Scripting will be integrated with several parts of the engine and game, including:

- Input
- Options
- Console input and output
- Unit specification
- Unit modifying

Scripting will be used in the input system to control the mappings from keyboard commands, like a button press, to gameplay instructions such as “move forward.” This will allow us to quickly and easily change the controls configuration as well as allow players to set up their own controls.

Units have many variables that define their behavior such as hit points, speed, and turn-rate. These will be specified in a Lua file for each unit. This will make it easy to determine the stats on each unit as well as quickly update them during play.

Options will be specified in a Lua file for easy parsing and modifying.

Units attributes will be updateable through a Lua interface, allowing for easy interactions with input, console system, AI, etc. This means you can control a unit using Lua exclusively.

Console input and output will be more complex than some of the other systems and will be responsible for displaying debug data and taking in user commands. Taking in commands will involve reading in a string of Lua code and then executing it. This means that anything we want to do with the console will need function hooks into the rest of the system. For example, we could increase player one's hit points by doing something like "Player[1].IncreaseHitPoints(10)" which would grab the first player and increase his hit points by 10.

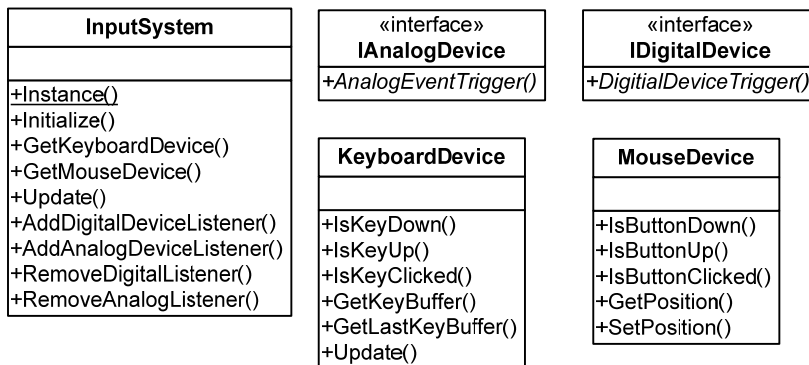
In order to do all of this, there will be one Lua state.

3.6 Input System

The input system will be mapped using a Lua script. This mapping can be updated at any time during the game. The input system will be a singleton that queries the keyboard, and mouse.

You must register your listener with the input system to receive events. The specific event you receive will be determined by the mapping. Any class can register / unregister with the input manager as long as it extends the appropriate interface.

The interface will have a virtual function that the listener class will need to define. This function will be called when an input event is triggered.



Properties of the input system will be

- Load from Lua – this will enable the input system to be setup based on a Lua file.
- Set device cooperation level – this will be set to active window only, active window and inactive window

3.7 Windowing System

The window system in the engine will give the caller direct access to a window, and directx device. This window will be what the game is drawn to, and creates the device to the hardware.

The border properties (MOVE, LOCK, SHOW_BAR) will be taken from the windows API, they restrict the way the window behaves and looks.

Window
+Window() +SetSize() +GetSize() +SetPosition() +GetPosition() +SetBorderProperties() +GetBorderProperties() +SetFullScreenState()

3.8 Render System

Rendering will happen in several stages. In the first stage, a scene is passed to the renderer. It takes this scene parses out the camera and lights and renders the elements of the scene to a render target, which it passes back.

Internal to this system, it will be using a deferred shading pipeline to render the color and lighting. This will be done with multiple passes and multiple render targets. Depth, color, and normals will be rendered to three targets in one pass, and then lighting will be calculated in a 2nd pass, and they will be combined in the third pass.

This can be done once, or several times. After all the scenes are drawn, one final combine call is made that takes in all previously rendered textures and brings them together, in order, for the final render. The GUI rendered target is passed in at this point and added on top of the 3D scene. These calls, render(scene) and renderCombine(renderTextures[]), are made by the state manager, and more specifically the active state.

3.8.1 Scene Management

Scene management entails three different goals: be able to update elements hierarchically for animation, organize elements spatially for fast frustum culling and collision detection, and sort based on render state for fast rendering. Instead of trying to create a monstrous data structure to do all three of these things, this engine will implement three different systems: a scene graph, a spatial graph, and a render state graph. In order to do this, three different entity classes will be used: a Spatial, a Node, and a Geometry. This three entity system is inspired by “3D Game Engine Design” by David Eberly. For more theory on this see Chapter 4.

Entities in this section will refer to a conceptual object that can be a spatial, node, or geometry, or a superset of them. A tank for example, would be referred to as an Entity as it is a grouping of Nodes, Spatials, and Geometry.

3.8.1.1 Spatial

Spatial will be the base class for all things existing within a scene. It has a position, rotation, and scale in relative and absolute coordinates as well as a bounding volume. It also has a parent Spatial. This allows it to be a leaf node in the scene graph. Note that there is no drawable element to a Spatial. It's possible that Spatial could be an interface.

Attributes:

- Relative Translation
- Relative Rotation
- Relative Scale
- Relative Matrix Transformation
- Absolute Matrix Transformation
- Parent
- BoundingBox
- Bool dirtyTransform

Functions

- Initialize()
- Activate()
- Deactivate()
- Update(dt) //updates itself and its children
- BuildMatrices //relative and absolute
- Translate(Vector3)
- Rotate(Vector3)
- Scale(Vector3)
- SetRelativePosition(Vector3)
- SetRelativeRotation(Vector3)
- SetRelativeScale(Vector3)
- GetRelativePosition(Vector3)
- GetRelativeRotation(Vector3)
- GetRelativeScale(Vector3)

3.8.1.2 Node

Nodes extend Spatial and add information that allows it to be a parent in a scene graph, such as a list of child Spatial. Note that a node's bounding volume will encompass the bounding volumes of its children. This will allow the passing of an entity's node from the scene graph to the spatial graph without having to pass and update each child. For something like a tank this makes sense. There is no need to do collision detection on each tread, the body, and the turret. We just want to test for collision against the whole thing.

Attributes:

- Children[]

Functions

- AddChild(Node*)

3.8.1.3 Geometry

Geometry extends Spatial and adds a draw-able component to it, allowing it to be a mesh, a primitive, or any other type of entity that needs to be displayed.

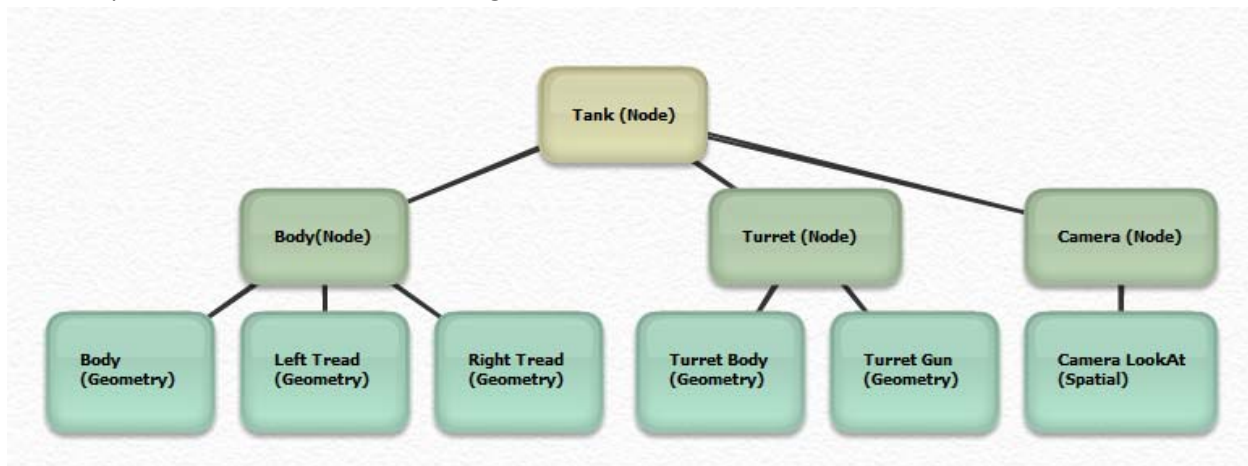
Attributes

- Mesh
- Material

Functions

- Render()
- SetMaterial()

An example of how these would work together in a Tank:



The update and render of all of this takes place in 6 steps.

1. Local Updates of Entities. (Player controls update the tank, AI updates NPCs)
2. Update Scene Graph to pack matrices
3. Update Spatial Graph to determine frustum culling

4. Sort Geometry for rendering
5. Render sorted Geometry

The local updating of entities happens outside of the render pipeline; it is only starting with the Scene Graph updating that concerns us.

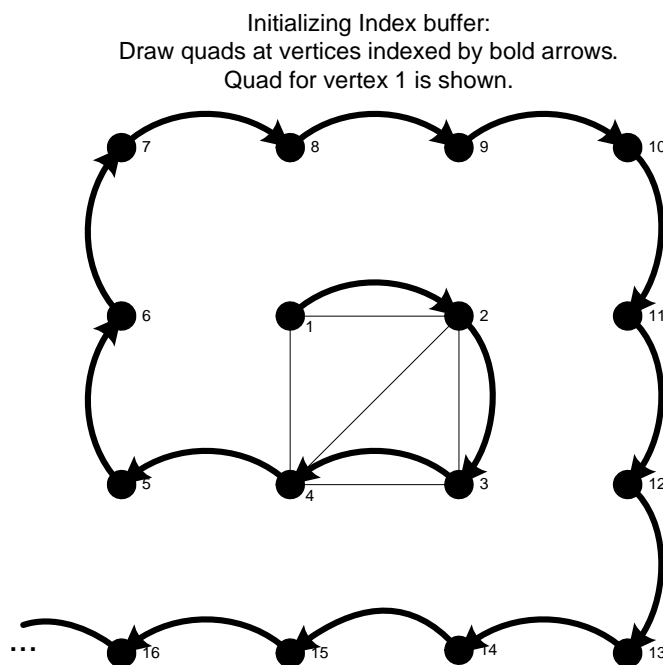
Types of Geometry objects:

- Mesh
- Terrain

3.8.1.4 Terrain

Since terrain is a special type of geometry, it will be handled differently in the Render System and Scene Graph. Terrain differs from general-purpose meshes in that it is defined by a height field texture, and that numerous optimizations can be made to its rendering procedure.

When passed to the GPU, terrain consists of a sheet of vertices arranged into a square plane, and an image representing a height field. The value of a pixel in the height field represents the height of the terrain at that location. Initially, the terrain vertices are created in a flat plane; during rendering they are moved into their correct positions by the terrain vertex shader.



In general, only a small portion of the terrain will be visible through the camera at any given time. Therefore, the portion of the terrain actually drawn can be limited to this window. This is accomplished by creating an initial set of vertices the same size as the maximum amount of terrain to be shown. The vertex buffer to contain these vertices should use a “swizzled walk” algorithm to address individual vertices. Instead of calculating the offset into an array in the usual way ($\text{offset} = x + y * \text{width}$), the bits of the x and y coordinates are interleaved in this manner:

$x_0y_0x_1y_1x_2y_2...$ up until the most significant bit of width . The purpose of this is to keep adjacent vertices closer together in the buffer, and increase vertex cache locality in the GPU. When initializing the index buffer, vertices should be indexed starting in the center, and moving outward in a spiral. This is so that smaller terrain squares can be rendered simply by passing a smaller square number to

DrawIndexed(). These optimizations impose some limits on the vertex mesh: the vertices must be laid out in a square that is power of 2 on each side.

Functions:

- Terrain(Heightmap, Mesh size): Terrain Constructor
- SyncCamera(Camera): Places the terrain vertices under this camera, and calculate the appropriate number of indices to render.
- Draw(): Render the Terrain

3.8.1.4.1 Decal Overlay

It is possible to dynamically overlay decal textures onto arbitrary positions of the terrain. These decals are drawn as quads with a specified texture applied. They are drawn to an off-screen buffer being used as a combined decal overlay during a first pass of the terrain. Only visible decals will be processed in this way, to reduce overhead. Also during the first pass, the previously rendered terrain is combined with the new decals, after aging by some amount. During the main draw pass, the combined decal texture is overlaid onto the terrain.

3.8.1.5 Sound Source

Sound Sources extend Spatial, and provide a way of representing 3D positional sound in the world.

Attributes:

- Sound reference

Functions:

- SetSound()

3.8.2 Scene Graph

Our scene will be organized using a scene graph in order to make hierarchical relations easier to modify and manage. For example, when a player moves their tank, they want both the tank body and the turret to move, so these will both be children of a general tank node.

A scene graph will have Spatial, Nodes, and Geometry in it. See previous section for an explanation of each of these. Every scene will have a root node that is the parent of all other spatial within the scene graph.

Types of nodes:

- Geometry
 - Mesh
 - Terrain
- Light
- Camera

- Particle Emitter
- Sound Source

SG Attributes:

- RootNode
- NodePools – holds the inactive nodes, one per type of node
 - NodePool[]
 - LightNodePool[]
 - TankNodePool[]
 - CameraNodePool[]
- ActiveCamera
- Lights

SG Functions

- Update()
- Render()
- Initialize()
- GetFreeNode() (returns a node from the node pool)
- GetRoot()

While we are going to want to store all of these in the hierarchical structure, certain types we will want to access quickly, such as the camera or the lights. Thus, these will also be stored in duplicate by reference in the scene graph. In addition to this, we will be storing a pool of unused spatial, nodes, and geometry for all unique types: friendly units, enemy units, player units, etc. This will prevent the instantiation and deletion of spatial at runtime.

If a system wants to add a new Node, it will request one from the SceneGraph, supplying a parent if relevant. It can then set any variables on this node that it needs. When it is done with the node, it informs the SceneGraph to delete it. The scene graph cleans the Node and puts it back into the appropriate Node pool.

Update takes the current scene, and traverses every dirty flagged entity and updates its absolute transformation matrix. After this is done, those dirty flagged entities will be passed to the Spatial Graph to have their positions updated and possibly moved within the spatial partitioning. As a producer-consumer system, this will be done by adding dirty flagged entities to the spatial graph input list after they have had their matrix packed.

3.8.2.1 Camera

One of the elements of the scene graph is a camera. A camera extends Node and will have position, orientation which can be changed by changing the camera's node values, but also perspective

information, point of view, aspect ratio, clipping planes. There can be one active camera per scene, although a scene may have any number of inactive cameras.

While what a camera sees will be rendered to a render target, the camera won't know anything about this.

Extra Camera Node Attributes

Camera
+SetLookAt() +SetEye() +SetUp() +SetView() +SetProjection() +GetView() +GetProjection()

NodeCamera
+LookAtNode()

FreeCamera
+Move() +Rotate()

3.8.2.2 Lights

We will have several types of lights in this engine:

- Ambient
- point lights
- directional
- spotlight

These can be attached to nodes within the scene graph and will also be registered as lights and will be passed to the shader during the lighting pass.

```
AddLight(Node* parent, LIGHT_TYPE type);
```

or

```
myNode->AddLight(LIGHT_TYPE type);
```

or both.

3.8.3 Spatial Graph

In order to speed up collision detection and prevent unnecessary rendering of objects outside of the frustum, the engine will be implementing a spatial partitioning system, or a spatial graph. Much of this implementation is based on "Real-Time Collision Detection" by Christer Ericson. For more on the theory behind this please see Chapter 7, "Spatial Partitioning."

There are several issues to consider when determining a spatial partitioning system for this game. We know that there will be a great number of immovable objects; trees, rocks, walls, buildings and these don't have to be updated. There will also be a large number of movable objects, which will almost entirely consist of tanks. And then there will be a large number of bullets that will be moving, but will also be contained in the area around the player and will not need to have collision tested on it (there is only the need to test bullets against objects already within the graph).

Several optimizations can be made based on this. A high resolution partition is only necessary around the player for determining collisions, a lower resolution partition can be used elsewhere and there is no need to detect collisions. A lower resolution grid will however need to be used elsewhere to determine what elements are within the frustum area, and thus should be on the high resolution grid.

One last thing to note is that all moving objects will be of relatively the same size, so no hierarchical partitioning is needed for those; however static objects may vary in size.

Thus three different partitioning systems will be used:

1. Static elements
2. Map-wide low resolution partition
3. Player localized high resolution partition

For the map-wide partition, there will be no collision detection happening, so the number of elements within each cell is irrelevant. It only matters if they are in the cells covered by the frustum; therefore the cell size will be approximately the same as the frustum and the frustum will cover four of these at a time.

Once we know which four of these cells intersect with the frustum, will take the elements within those four cells and add them to the high-resolution partition. This partition is used to determine collisions between tanks, bullets and tanks, tanks and static elements, and for more precise frustum culling. Therefore, the cells of this partition will be tank sized. Elements won't be moving in and out of the high resolution grid very often, so they will remain in the grid and be updated as needed.

The exact implementation of these partitioning systems is still undecided, however, expect to use: Implicit Grid, Grid as Array, or a Hash implementation. Because of the lack of size variability there is no need for something as complex as a hierarchical tree or a quad-tree.

Updating takes place as follows: the scene graph packs all of its matrices and then passes the updated ones to the spatial graph using its input queue. Note that not all entities will need to be passed. Stuff like a camera has no place in the spatial graph, nor do entity sub elements like a tank's turret. Thus, Nodes will have to be registered with the spatial graph and only those will be passed to it.

As output, the spatial graph will parse the un-culled elements and pass their Geometry children. No need to pass Nodes as they have no draw-able elements. In our tank example, the tank, body, and turret nodes would not get passed, but all of the geometry would.

Attributes

- InputQueue[]

Functions

- Update

3.8.4 Render Graph

After the scene graph has updated and after the spatial graph has determined visibility, the render graph takes over. It has its own input list populated by the spatial graph which it can now sort and order as much as it wants. After this is done, it then loops over the list of drawables and draws them, ending our render loop. Right now, after looking at data concerning render state sorting, we are placing a very low priority on doing this as the improvements will be minimal.

http://aras-p.info/texts/d3dx_fx_states.html Has a very nice article showing the speedups to be gained by sorting based on state changes.

Attributes

- InputList[]

Functions

- Sort
- Render

3.8.5 Render Targets

The rendering pipeline will make extensive use of multiple render targets (MRT) in order to achieve certain effects, as well as to accommodate various debug features, and integrate the GUI with the 3D scene. We would like to make this process dynamic and scriptable, but this is still being looked into.

The use of multiple render targets during render will be hidden within the rendering system. The only output you get will be the final image rendered to a texture.

It will be nice to have access to several of the render targets used in the rendering process, such as normals, depth, shading, etc. These will only exist with the renderer, but will have accessors so that they can be used elsewhere; debug drawing for example.

- GetDepthTexture();

- GetNormalTexture();
- GetColorTexture();
- GetShadingTexture();

3.8.6 Bounding Box

We will use an object aligned bounding box implementation based on “Real-Time Collision Detection” by Christer Ericson. A bounding box will need to be given a pair of X, Y values, upon creation, specifying the minimum and maximum bounds of the object. They also have the ability to be resized post creation. Bounding boxes will have the ability to update based on a transformation given. Collision boxes will have the functionality to check collision against other bounding volumes.

For debugging purposes the bounding box will have the ability to be rendered. Initialize render will set up the vertex buffer with information for rendering the box. Update render will update the vertex buffer based on the current bounding box.

BoundingBox
-BaseBox[5] -CurrBox[5]
+Signed2DTriArea() +Test2DSegmentSegment() +SetBounds() +UpdateBounds() +Intersect() -InitializerRender() +UpdateRender() +Render()

3.8.7 Particle System

Particle systems will be generated on the GPU through a two phase process using shaders. Each phase is an individual technique in HLSL with a single pass. The first step of the shader process passes the vertices from the vertex shader straight to the geometry shader. If the current vertex is an emitter new particles are created and appended to the output. If a particle has reached the end of its life then it will not be appended to the output list.

The Geometry shader is set to stream out to a new buffer. This buffer contains vertices which describe the positions of individual particles in a system. The second vertex shader takes each of these vertices and calculates their new position based on the physics equation: $\text{currentPosition} = \text{initialPosition} + \text{initialVelocity} * \text{time} + 0.5 * \text{acceleration} * \text{time}^2$. Opacity of a given particle is also calculated based on interpolation between a start and end opacity at a given time (HLSL smoothstep function).

Next the second geometry shader will take the positions of each particle in the system and generate a 2D sprite based on the desired sprite size and current location. Texture coordinates will also be assigned to the new vertices at this point. Texture coordinates will be based on index values into the sprite sheet (based on width and height) based on time elapsed till depth.

Part of a particle system is creating effects which have some randomness to them in order to seem more natural. As HLSL does not have a built in random number generator a texture will need to be generated CPU side containing a large number of pre-calculated random numbers (as RGB values) to be used in the shader for random numbers. An RGB value can be expressed as a 3D vector in HLSL.

3.8.7.1 Particle System Description

The particle system description is a struct which describes a single particle system. This allows the designer to create multiple particle systems easily with the same look and feel. It is a stored argument list to define a single particle system which will be placed into the world.

- emitterLife – Defines the length of time in seconds which this system will create new particles. Once this time has ended the system will no longer create new particles for the system. If the emitters life has passed and all particles in the system have passed their life span then the system is dead.
- emitRate – Defines how many particles per second the emitter creates
- particleLife – Defines the maximum length of time in seconds which an individual particle in the system will stay in the system. Individual particles will receive life span between 0.0 and the maximum particle life. Once this time has ended the particle will be removed from the system.
- particleScaleStart – Defines the scale multiplier when creating the sprites to draw particles in a system. Particles will initially start off at this scale calculated from a 1.0 by 1.0 sprite. So starting at scale = 0.1 will set the particles at 0.1 by 0.1.
- particleScaleEnd – Defines the scale multiplier when creating the sprites to draw particles in a system. Particles will end at this scale calculated from a 1.0 by 1.0 sprite. Ending at scale = 2.0 will set the particle at 2.0 by 2.0.
- maxParticles – Defines the maximum number of particles which are allowed to be alive in a single particle system. The system will not create any new particles unless the current number of particles in the system is less than this number.
- system name/id – An identifier for the particle system description.
- texture name/id – An identifier for the texture which this particle system uses to render individual particles.
- Num sprites width – The number of sprites in the textures width
- Num sprites height – The number of sprites in the textures height
- effect name/id – An identifier for the effect which is used when rendering this particle system.
- number of linked systems – Size of the linked system descriptions array (below)
- linked system descriptions –An array of other particle system descriptions which should be created at the same time of this description.

3.8.7.2 Particle Manager

The particle system manager (PSM) will pre-allocate a number of particle systems to be used during game play. This prevents from allocating memory for systems during game play and creating potential slowdowns. The initialize function of the particle manager will take care of creating a set number of systems for use. It will take in the number of systems as a variable. During initialization of the particle manager the random texture will be generated to be passed to individual particle systems.

The PSM will take in particle system descriptions. These descriptions will be stored in an array to be used when activating a particle system. This allows for system types to be preloaded and stored only once for use with more than one system.

The PSM will also be able to retrieve a particle system description from the array based on the descriptions name/identification. When adding a particle system to the world the description will need to be retrieved from the array. The description, if found, is returned by the function. If the description is not found a value of zero will be returned.

In order to create a particle system in the world an add system function will need to be called. This will take in a particle description name/id, emitter position and direction. If the particle description is found the system will be added to the active list, given there are inactive systems available, or return 0 if the system cannot be created. If a description has additional systems linked then the add system will be called as needed to create those systems as well.

Linked systems will all receive the same position and direction of the system they are under. Each will fire at the same time as the first system. If systems need to be offset in time, position or direction they will need to be added separately. An example of a linked system would be creating an explosion which hits something metal creating sparks as well.

Update will loop through the list of currently active particle systems and call each of their update functions individually. As the particle systems each need the delta time and run time this function will require both as well in order to pass on the correct information. If a particle system returns false then the system is moved from the active list back to the inactive list.

Render will loop through the list of currently active particle system and call each of their render functions individually.

GetLights will loop through the individual particle systems in the manager and converting their particles to light sources. An array of lights will be generated and counted. This will set the outLights and numLightsOut variables.

3.8.7.2.1 Functions

- Void Initialize(int size)
- Void AddDescription(ParticleSystemDescription desc)
- Void GetDescription(string name or int id)

- AddSystem(string name, Vector3 position, Vector3 direction)
- Update(float deltaTime)
- Render()
- Void GetLights(Light* outLights, int& numLightsOut)

3.8.7.2.2 Variables

- Array<ParticleSystem> activeList
- Array<ParticleSystem> inactiveList
- Array<ParticleSystemDescription> descriptionList
- TextureResourceView randomTexture

3.8.7.3 Particle System (Class)

A particle system is the class which updates and renders a single particle system based on a particle system description.

In order to set up a particle system for use it will first need to be initialized. When initializing a particle system it will need a particle system description, emitter location, emitter direction, and random texture. A delay may also be given to the particle system although it is preset to 0.0 by default. Initialize will create the initial vertex buffer and create the appropriate shader variable assessors on the CPU side.

Upon updating a particle system the change in time between the last update will need to be passed in. This allows for the particle system to calculate new positions based on time. The delta time will also be added to the run time in order to help with indexing into the random number texture. If the run time of a system is larger than the emitter life + particle life then the function will return false in order to be marked as inactive in the particle manager.

Render will set variables on the shader for each run. The view and projection matrices will be given, the delta time, game time, and information from the particle system description. On the first run of render the initial vertex buffer will be used, subsequent runs will use the secondary vertex buffer. A stream out vertex buffer will be used in order to grab output from the geometry shader in the first technique. This will contain the previous particle system state plus any additional particles created.

Vertices going in and out of the first geometry shader are as a point list. Going into the second geometry shader the vertices are a point list but the output will be as a triangle list.

Once this phase of render is complete the vertex buffer (initial or secondary) is swapped with the stream out vertex buffer. This is used to draw each of the particles in the system with the second technique defined in the shader.

3.8.7.3.1 Functions

- Void Initialize(ParticleSystemDesc desc, Vector3 position, Vector3 dir, TextureResourceView random, float delay = 0.0)

- bool Update(float deltaTime)
- Void Render()

3.8.7.3.2 Variables

- float runTime – allows for runTime to be shared between Update() and Render()
- float deltaTime - allows for deltaTime to be shared between Update() and Render()
- float age – keeps track of the emitters age CPU side in order to tell when the system has stopped emitting
- initial vertex buffer – initial state of the particle system
- draw vertex buffer – secondary vertex buffer, used after the first run instead of initial vertex buffer
- stream out vertex buffer – stores the output of the first geometry shader
- texture resource view – shader resource for the draw texture
- random texture resource view – shader resource for the random number texture
- view projection matrix variable – stores the multiplication of the cameras view and projection matrix for use in the shader
- delta time variable – stores the delta time for use in the shader
- run time variable – stores the run time for use in the shader
- emit life variable – stores the emitter max life for the shader
- particle life variable – stores the particle max life for the shader
- particle start scale variable – stores the particle draw scale for the shader
- particle end scale variable – stores the particle draw scale for the shader
- sprite width var
- sprite height var
- max particle variable – stores the max number of particles for the shader

3.8.7.4 System Types

Here will be a list of the various particle systems to be incorporated into our game. What the variables will be set to in a particle system description as well as the look and purpose of the individual system.

- Explosion
- Spark (bullet hitting metal)
- Water explosion
- Snow Explosion
- Gun firing
- Smoke
- Snow
- Shield Hit

3.8.8 Other Effects

3.8.8.1 Foliage

In order to make certain parts of the terrain look less empty, the engine will make use of instanced, billboarded foliage.

3.9 Resource System

Several types of resources will be used in the Marabunta engine including: textures, 3D models, audio, shaders. These will be managed by several resource management systems that will be globally available to the entire engine.

3.9.1.1 Mesh File Format

The mesh file format is a simple binary format. The primary purpose for having a custom file format is to ease the transition from a modeling program to loading in the game. The secondary purpose for this is to have total control over what we want to export and import into the game so we can have a quick loading sequence.

The format is broken down into two files. The first file is the model file; this file contains information for building a static model. The information required in this file is the vertex position, texture coordinates, and normals. To parse this file you need to read in the header using the CFHeader struct. Then you can begin to read in the vertex information, and finally the index buffer information.

File format is defined as follows

- Header Struct(12 bytes)
 - (int) Ident – a code to verify the file type
 - (int) Version – the version of the file type
 - (int) number of vertices in the file
 - (int) number of indices in the file
- Vertex Information Struct (32 Bytes) * number of vertices
 - 12 Bytes (X, Y, Z as floats) Vertices
 - 12 bytes (X, Y, Z as floats) Normals
 - 8 bytes (U, V as floats) UV Coordinates
- 4 Bytes (index as unsigned int) * number of indices in file

The second file is an animation file. This file contains the joint, and weight information for each of the vertex for animating the model. **This file format is a work in progress, and may change to better suit the needs of development.** Similar to the model file, to read this file first you must read the header, then read the file information

Animation File format

- Header (4 Bytes)

- (int) number of joints
- Joint (56 Bytes)
 - (CFVertex [12 Bytes] X,Y,Z as floats) Position
 - (CFVertex [12 Bytes] X,Y,Z as floats) Rotation
 - (CFVertex [12 Bytes] X,Y,Z as floats) Orientation
 - (CFVertex [12 Bytes] X,Y,Z as floats) Scale
 - (int) Number of Weights
 - (int)Number of Key Frames
- Per joint
 - 36 Bytes (KeyFrame) * Number of Key Frames
 - 4 Bytes (weights as floats) * Number of Weights

3.9.1.2 Animator

The animator is a class that is responsible for loading animated meshes. This class will setup a skeleton structure that has a single parent joint. Each joint can have n children. The animator class will support the functionality to animate a mesh based on keyframe sets.

Animator
+LoadMesh() +Update() +Render() +AddKeyframeSequence() +PlaySequence() +SetFrame() +GetSkeleton()

- AddKeyframeSequence – Accepts a string that is the name of the sequence, a start frame, end frame, and time of traversal. Eg.. “Fire”, 10,20, 1000 (time in milliseconds)
- PlaySequence – accepts the string name of the sequence eg.. “Play”

3.9.1.2.1 Skeleton

The skeleton class holds a linear list of joints, and a parent joint that holds the joint hierarchy. The update method is responsible for updating the parent joint and setting the appropriate frame on the parent.

Skeleton
+Skeleton() +Setup() +Update() +MoveJoint() +RotateJoint() +ScaleJoint() +GetJoint() +GetJoints()

3.9.1.2.2 Joint

Joint is a class that manages the matrixes that are used for animation of a model. Animation can be controlled using the setframe method or explicitly calling move,rotate,scale, and set methods.

Joint
+SetPosition() +SetRotation() +SetScale() +Move() +Rotate() +Scale() +SetFrame() +Update() +UpdateMatrix() +Render() +AddChild()

3.9.1.3 Mesh

The mesh class is responsible for loading static mesh objects. This class is to be used as a child of a node and there for does not have move,rotate, and scale methods.

Mesh
+Load() +Update() +Render() +()

3.9.1.4 Texture Manager

The texture manager will store an array of texture resource structures. The manager will have the ability to load a texture from file, create a new texture in memory as well as retrieve a texture resource from the list.

Load texture will take in a filename to load the texture from the file system. If the filename already exists in the texture resource array then the texture will not be loaded a second time. The filename will be the files relative path. Once the file is loaded the system will store the texture resource view and

name in the array. The index of the texture resource will be returned when added or found to be previously loaded.

The manager will have the ability to create an empty texture for use in the engine. In order to create a new texture a height, width and format for the texture will need to be given. Based on need this function could be expanded to create texture arrays, mip-levels, etc.

In order to retrieve a texture from the array there will be a retrieve texture function which takes the index of a given texture into the array. If the index is valid the texture resource view will be returned. If the index is not valid then zero will be returned

3.9.1.4.1 Functions

- int LoadTexture(filename)
- int CreateTexture(width, height, format)
- void RetrieveTexture(index, TextureResourceView out)

3.9.1.4.2 Variables

- Array<TextureResource> resourceList

3.9.2 Shader Manager

The shader manager will be responsible for loading and caching shaders in the engine. This class is here to assist the loading and unloading of effect objects. It will be used in much the same way the texture manager is.

The manager will have a list of effects that have already been loaded in the system. These effects can be retrieved using functions that take in the effect name as an argument.

To load a new effect you call the load method with the path to the effect, this path will also be the key to retrieve the effect at a later point.

3.9.2.1.1 Functions

- ID3D10Effect* LoadEffect(filename)
- void RetrieveEffect(filename, ID3D10Effect** out)

3.9.2.1.2 Variables

- Array<ID3D10Effect*> resourceList

3.9.3 Audio Files

3.10 Console System

A console system using Lua will be built into the engine. This will allow a user or developer to pull up a console, write Lua commands into the console, and modify the game as it is running.

The console will be brought up when the user presses “~” and will cover most or all of the screen, dimming what is going on in the background. The user can then execute commands in the console to modify the state of the game. Pressing “~” again will close the console.

3.11 Material System

The engine will make use of a dynamic material system. This will allow us to attach materials to renderable objects and render them appropriately.

3.12 Debugging System

Debugging system will have the ability to log messages based on their message level. Instead of blocking out all debugging messages in release mode this allows us to turn them on and off based on message level. Levels lower than the current level will be output as well. An example is that we only want to see critical debugging messages during release mode but when debugging we will want to see varying levels of messages.

The debugging system can be set to output data to a separate console window, text file or in game console window. Add output places a new output form into the output list for use when printing a message. If the output type is already in the list then it will not be added. Remove output allows for the output method to be removed from the output list for use when printing a message.

Settings in the debugging system such as the level to print debugging messages can be set in either the in game console and file based game settings. Set level allows for this functionality to occur.

Text file debugging logs are stored in the log directory with filename based on the current date. Logs will append to one another based on date. The log of a given run will start with a line stating it is an instance of the system and the time it started.

In order to place a message in the debugger for printing the print message function need to be called. This function requires the debugging message and debugging level on which to start showing the specific message.

3.12.1.1 Functions

- Debugger(int level)
- SetLevel(int level)
- AddOutput(enum OutputType)
- RemoveOutput(enum OutputType)
- PrintMessage(string Message, int level)

3.12.1.2 Variables

- int currentlevel
- Array<unknown type> outputList

3.12.1.3 Log format

[01/01/2009 5:45:13] System:Class:Function "Error Doing something"

3.13 Timer System

This class will provide the ability to create timers. A timer will have the ability to start, stop, get change in time and reset.

Timer
-StartTime -EndTime -LastTime
+Timer() +Start() +Stop() +GetDT() +Reset()

4 Game Technology

This section describes the technical aspects of our game. Does not talk about the engine, but might talk about how our game utilizes it.

4.1 Main

The main class is one that initializes the engine, state system, and networking. This class will have an update, render cycle. The update cycle will call update on the network, then call update on the state system, and finally update on the engine. After the update cycle finishes the loop will then check the dt and determine if a render cycle needs to be called. The render cycle will call render on the state system and finally the engine.

4.2 Tank Managers

Each type of tank will have a manger so we don't new any classes during the game loop. The tank managers will be responsible for keeping a queue of tanks and rendering only the ones that are active. A single manager may be able to manage all the tanks as they will be instanced.

4.3

4.4 State System

Our game will be organized into several different states such as: menu state, play state, etc. Different states might make use of different rendering pipelines. There will be a state manager that is responsible for remembering what the current state is, and switching between states. This state manager also tells each state to update and render.

- AddState – This function adds a state to the state system, takes in a state and returns an index
- RemoveState – Removes a state from the state system, referred to by index or state pointer
- ActivateState – this function will activate a state based on id or a pointer to the state you want to active, on the next update cycle the state machine will call deactivate on the current state and activate on this state.
- Update – updates the states and handles activate, and deactivate calls.
- Render – calls render on the states.

StateSystem
-inactiveStates
-activeState
+AddState()
+RemoveState()
+ActivateState()
+Update()
+Render()
+Initialize()

4.5 IState

Any states that are to be used by the state system will need to implement this interface. The interface defines several functions that are needed by the state system

- Initialize – This gets called one time when the state machine gets initialized
- Activated – This gets called every time the state becomes activated
- Deactivated – This gets called every time the state becomes deactivated
- Update – this gets called every frame.
- Render – This gets called every draw call.

IState
+Initialize()
+Activated()
+Deactivated()
+Update()
+Render()

4.5.1 Logo State

Responsible for presenting the team logo.

Entry Points: Game startup

Exits To: Main Menu State

Introduction Transition

Fade from black to logo in 0.5 seconds

Exit Transition

Fade from logo to black in 0.5 seconds

Interaction

Pressing any key will skip the introduction transition and exit transition and move to the introduction transition of the Main Menu State assuming all of the assets for the Menus are loaded.

Responsibilities

Responsible for loading all of the assets for the main menu state

Required Assets

Game Logo

4.5.2 Menu State

Where the menu action goes on.

Entry Points: Logo State

Exits To: Options State, Team Game State, Solo Game State, Credits

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

pressing the move menu item button will change the item selection up and down. When the activate item button is pressed the selected item triggers a state change.

Responsibilities

Responsible for displaying the start menu.

Required Assets**4.5.3 Credits State**

Entry Points: Main Menu State

Exits To: Quits the program, Main Menu

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

The back button will bring the user back to the main menu. The action button will exit the game.

Responsibilities

Responsible for displaying the credits

Required Assets

Credits screen

4.5.4 Options State

Where the options can be changed

Entry Points: Main menu

Exits To: Main menu

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

pressing the move menu item button will change the item selection up and down. When the activate item button is pressed the selected item triggers a change in the options.

Responsibilities

Responsible for displaying the start menu.

Required Assets

4.5.5 Team Game Selection State

Where the menu action goes on.

Entry Points: Logo State

Exits To: Options State, Team Game State, Solo Game State, Credits

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

pressing the move menu item button will change the item selection up and down. When the activate item button is pressed the selected item triggers a state change.

Responsibilities

Responsible for displaying the start menu.

Required Assets

4.5.6 Team Game Setup State

Where the menu action goes on.

Entry Points: Logo State

Exits To: Options State, Team Game State, Solo Game State, Credits

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

pressing the move menu item button will change the item selection up and down. When the activate item button is pressed the selected item triggers a state change.

Responsibilities

Responsible for displaying the start menu.

Required Assets

4.5.7 Solo Game Setup State

Where the menu action goes on.

Entry Points: Logo State

Exits To: Options State, Team Game State, Solo Game State, Credits

Introduction Transition

Fade from black to screen in 0.5 seconds

Exit Transition

Fade from screen to black in 0.5 seconds

Interaction

pressing the move menu item button will change the item selection up and down. When the activate item button is pressed the selected item triggers a state change.

Responsibilities

Responsible for displaying the start menu.

Required Assets

4.5.8 Load State

Purpose of the load state is to load the content from disk required for the play state. The load state parse the mission file and load content to managers as needed. In order to speed up the processing of loading data and maintaining a visual representation while loading this state will utilize a threading system.

Entry Points: Solo Game State, Team Game Setup

Exits To: Game

Introduction Transition:

Exit Transition:

Interaction

None

Responsibilities

Handle loading all content required for the Game state

Required Assets

4.5.8.1 Load Thread

The load thread will lock on the load queue and dequeue up to the number of elements described in the load args received as input. If the queue no longer has elements the thread will exit. A switch statement will direct point each of the load resources to the correct manager for loading content.

In case the program exits during the loading process the terminate event will be signaled in order to exit the thread cleanly before continuing to load the rest of the content.

4.5.8.2 Load Args

The load args structure will contain a pointer to the load queue as well as an unsigned integer greater than 0 to define how many items the load thread should dequeue per iteration. The load args will also contain an event for termination.

4.5.8.3 Load Queue

The load queue is shared memory which will be accessed by individual load threads. This structure contains a STL queue as well as a CRITICAL_SECTION in order to lock on queue and dequeue actions.

4.5.8.4 Load Resource

The load resource is a structure containing a filename string of content to be loaded as well as an integer value (set by an enum) describing the system used to load the file.

Enum values: Texture, Mesh, Sound, Music, Shader, Configuration (XML), Terrain, Level Data

4.5.9 Play State

This is where all the tank action happens.

4.6 Influence Map

An influence map is an item that represents the threat level associated with a given location on the terrain. This map is the same size as the terrain and updates as each unit moves across the environment. Each object will have a threat level associated with it. Friendly units and objects will have a positive threat level and enemy units will have a negative threat level. Only enemy units that are visible to friendly units will be taken into account when updating the influence map.

Threat levels get updated as a radial function that distributes evenly away from the center point. A unit with a threat level of 10 will be in the center of this radius with each tile around him a 9 and each tile around the 9's an 8 etc...

The influence map will need to have a scale factor that scales it down from the high resolution of the terrain to a manageable resolution of for units to update it real time.

4.7 AI System

The AI is broken into two sections, High Level and Low Level. There is only one high level AI instance per side, while each NPC unit has its own low level AI. The high level AI analyzes the game state and sends commands to the low level AI. The low level AI is embodied in the NPC Controller class, instances of which can be set as controllers of Unit instances.

4.7.1 High Level Decision Making

A high level architecture has been designed to manage the AI units as a whole. This system will periodically analyze the game, and decided what units / groups of units should do next as part of a high level strategy.

To accomplish this, the system uses a pre-trained neural network to make these decisions. Below is described about the various inputs / outputs to the system and how they are used to train the system. The goal of the neural network is to extract the relevant information from a variety of inputs and determine the strategy for a cluster of units.

4.7.1.1 Input

The input to the system will be a large area of weights from the influence map. This will centered on a cluster of tanks but will be larger than the sight of the tanks. This will require at least 100 inputs for the neural network.

The next set of inputs will be features of the map. These features include, number of friendly units in the cluster, and number of enemy units in/around the cluster.

4.7.1.2 Outputs

The outputs from the network will be used to determine a decision tree for a cluster of units to use. There are four decision trees in total that are, Attack and Regroup, Attack and Disperse, Defend and Regroup, and Defend and Disperse. The next set of outputs is an X, Y coordinate that is to be used as a direction for the cluster to head in.

An attack and regroup cluster will move to a location and stick together as a group or in a formation. While an attack and disperse cluster will attempt to ambush a location. The defend and regroup cluster will defend a building or unit at that location and stay in a tight group. A defend and disperse cluster will move in a general direction to that point and split up. They may not re-gather on a single point.

The outputs of the system will be

- Attack / Defend (-1 / 1) – This will be a fuzzy value between -1 and 1.
- Regroup / Disperse (-1 / 1) – This will also be a fuzzy value between -1 for regroup and 1 for disperse.
- X – This will be a normalized value between -1 and 1. The range will represent a tile in the set of inputs for the units to go. This value will need some post processing that will convert it into world space coordinates
- Y (Heading) – This is the same as the X value but in the Y coordinate.

4.7.1.3 Training

The network will be pre-trained to a specific percentage. The network will also learn using a fitness function through the execution of the game. The training set will be determined by hand and will be large enough to encompass a large set of situations.

We do not need to worry about over-fitting the data at this point because the network will not be trained to have the lowest error rate. Instead the network will be trained to have a low error rate to leave some variability for in game learning to occur.

4.7.1.4 Fitness Function

The fitness function that will be used to train the network will be based off a rules based system. The rules will be based off research of strategies used in AI for RTS games.

The determination of outputs will be done by analyzing the inputs to the system along with additional unit information to calculate an appropriate output response.

4.7.2 Low-Level Behaviors

Each NPC unit shall have a set of 4 behavior trees that govern its behavior. These trees correspond to the directives Attack and Regroup, Attack and Disperse, Defend and Regroup, and Defend and Disperse as stated above. Each unit type must have all four trees defined; unit types are allowed to share tree definitions. Trees are defined either in XML or Lua, whichever would be easier considering their hierarchical nature and the rest of the asset loading system.

Behavior trees shall be composed of derivatives of the abstract Behavior Tree Node class. All behavior tree nodes have at least two methods: execute and plan. Plan is used during the planning phase of the algorithm, and returns the estimated cost of executing the node. Execute actually performs the behavior specified by the node.

Each NPC controller instance maintains references to the behavior trees, as well as the currently active tree, and currently active node. The state of the NPC should be kept entirely within the NPC controller so that multiple NPCs can share the same behavior trees. In addition to the behavior tree itself all NPC controllers have a node stack, an input stack and plan objects.

The node stack is used to represent the current behavior of the NPC, along with the behaviors taken to reach the current one. When the NPC is initialized, the root node is pushed on the stack. When an aggregate node (a node with children) is executed, it may push additional nodes onto the stack. The top of the stack is always assumed to be the current behavior to execute. The execute method returns a status, one of Failure, Pending or Success. On Failure or Success, the top node is popped off the node stack (unless it is the root). On Pending, control remains with the top node, which may have changed.

4.7.2.1 Behavior Tree Nodes

These nodes are available to be used in the definition of behavior trees:

- Move/Shoot
 - Moves along path and shoots at current target if in range
- Move/Shoot as Group
 - Same as Move/Shoot, but using group movement behavior
- Follow/Shoot with Player unit
 - Follow an ally Player unit while shooting at current target
- Target Heading
 - Sets target to heading
- Target Unit
 - Sets target to enemy unit (will track target)
- Path to Cover
 - Finds nearest suitable cover and plots path
- Path to Location
 - Plots shortest path to global location
- Path to Heading
 - Plots shortest path to relative location
- Path to Base
 - Plots shortest path to friendly base
- Join Group
 - Joins a group specified by unit type (flock/formation)
- Disband Group
 - Leave current group

- Sequence
 - Execute all child nodes in sequence until one fails
- Iterative/Plain Selector
 - Execute all child nodes in sequence until one succeeds
- Learning Selector
 - Execute all child nodes in sequence until one succeeds
 - Move succeeding child node to front of list
- Planning Selector
 - Selects child node to execute during plan phase
 - Core of behavior tree planning
- Random Selector
 - Randomly selects child nodes to execute
 - More academic than actually useful
- Script Node
 - Asynchronously executes a Lua script – returns Pending while the script is running
 - Lua script returns a binary value: false = failure, true = success

4.7.2.2 Input Events

The input stack is a way to customize nodes with early termination conditions. When a node is pushed onto the node stack, the input events associated with it are pushed onto the input stack. Before a node is executed, the entire input stack is checked from top to bottom to see if any of the input conditions are true. If one is, the current node is terminated as if it had returned the result condition specified in the input event.

Input events are specified as part of the behavior tree, but as properties of nodes, not nodes themselves. However if an XML format is used, Input events should be XML elements within their containing node, not attributes. Along with their trigger condition, specified by their type, input events carry a result to supply when triggered: Success or Failure.

Possible input events are listed below. Unless specified, all events are limited to the unit's own range of vision/fire.

- Enemy in range
- Target in range
- Under Fire(both hits and misses)
- Ally Under Fire
- Ally Attacking Enemy
- Ally Player Arrived
- Enemy Player Arrived
- Base/Objective Under Attack

4.7.2.3 Planning

Finally, the NPC controller's plan object represents a particular path through the tree. The NPC controller enters the planning phase whenever the plan is empty, or has been invalidated by a failed behavior. This is controlled by Planning Selector nodes in the behavior tree when they are executed. When a Planning Selector enters the plan phase, it queries all of its children for their estimated cost of executing their behavior. It then records the lowest cost child as the next node to execute in the plan, and pushes it onto the node stack. Aggregate nodes must also report a plan cost, as appropriate for their behavior. If all children of a Planning Selector report an infinite cost – meaning they are not applicable to the current situation – the failed behavior signal is propagated to the selector node's parent. When Plan is called on a Planning Selector node – meaning one of its parent selectors is planning – it must record its decision in the controller's plan object, but not push any nodes onto the node stack.

4.8 Threading for AI

This section describes the classes, structures and interfaces which are used for multi-threading the AI described above.

4.8.1 Thread

A thread is created using the Win32 API and is given a static function pointer as well as arguments to be taken in. Threads will be created based on a function which will continuously loop until asked to either sleep or terminate based on an event passed in as part of the argument passed in upon creation

4.8.2 Thread Arguments

The thread argument structure is used one per thread. The argument holds pointers for the sleep, wake and terminate events. It also is given a granularity value which determines how much work the thread is allowed to complete per iteration. Thread arguments will also contain a pointer to the task manager for the thread to use.

4.8.3 Thread Pool

The thread pool class upon creation will determine how many processing cores are available on the current machine. It will then create N-1, where N is the number of available processing cores, as a thread pool. The thread pool will control thread creation, termination, sleeping, waking, and passing of arguments to the threads.

4.8.4 Task Manager

The task manager implements a shared memory interface and allows systems to add and retrieve tasks for use by threads. A thread requests work from the task manager by giving it a maximum granularity value and then receives back work based on how much it can accept.

4.8.5 Task Interface

The task interface is implemented by classes in order for a thread to process what is defined in the update function. Tasks are assigned a cost such that when requested by the task manager they are able to determine if they should be assigned or not.

4.8.6 Shared Memory Interface

This interface is implemented by classes which will need to be protected by locking before memory is written to or read from. It has the ability to lock, clean up the lock, initialize the lock and unlock.

4.8.7 Double Buffer Interface

Double buffer is an extension of shared memory. It provides the functionality for creating double buffered classes. It will be able to write to the back buffer, retrieve from the front buffer and swap. Locks will be used to prevent the buffer from being swapped during reads and writes to the buffered data.

4.9 Unit

A unit is an active entity in the game, such as tanks or turrets. These may or may not be mobile, but are able to actively affect the game environment; they will have hit points and be destructible. Units are controlled by either the player or the low-level AI system. Unit types are defined by their parameters, which are specified through Lua scripts. These attributes are listed in the Attributes section below.

4.9.1 UnitManager

The Unit Manager creates a set of units of each type on load; one each of the player classes per player, and a set number of NPC units. When requested by the game, they are simply linked into the simulation. On death, NPC units are returned to the not-in-use pool. Player units are immediately recycled for respawn.

4.9.2 Attributes

4.9.2.1 Unit Class Members

Since player and non-player unit types will be defined by Lua script, a single Unit class is necessary. The interface to this class is shown on the right. The state of any particular unit is defined by the class' protected (#) members, namely those below. When the unit is spawned, these values are set.

- Shields – Unit's current shields. Unit dies when this reaches 0.
- Energy – Used as ammunition for Player unit weapons.
- Position – Current world position of the unit.
- Speed – Current forward speed of the unit.
- Rotation – Current world rotation of the unit.
- Rotation Speed – Current world rotational velocity of the unit.
- Turret Rotation – Current world rotation of the turret.
- Team – Current team the unit belongs to.

Unit
#Shields
#Energy
#Position
#Speed
#Rotation
#Rotation Speed
#Turret Rotation
#Team
+Name
+Controller
+Max Speed
+Max Turn Speed
+ Turret Turn Speed
+Acceleration
+Max Shields
+Shield Regeneration
Rate
+Energy Regeneration
Rate
+FOV
+Primary Weapon
+Secondary Weapon
+Shield Type
+Base Mesh
+Base Texture
+Base Destroyed
Texture
+Turret Mesh
+Turret Texture
+Turret Destroyed
Texture
+Update()
+Initialize()
+Damage()
+Accelerate()
+Rotate()
+Aim Direction()
+Deploy()
+Fire Primary()
+Fire Secondary()

4.9.2.2 Unit Parameters

Unit parameters are values set by script that govern the operation of the unit. These are set when the script is loaded and aren't changed during the course of gameplay (fields marked with a "+" above).

- Name – The Unit's type, in human-readable form.
- Controller – Unit's controller, either Player or AI Controller.
- Max Speed – Maximum speed the unit can travel at.
- Max Turn Speed – Maximum rate unit can turn at.
- Turret Turn Speed – Rate at which the turret rotates.
- Acceleration – Amount the unit accelerates at.
- Max Shields – Maximum shields the unit is allowed to have.
- Shield Regeneration Rate – Rate at which the unit's shields are regained.
- Energy Regeneration Rate – Rate at which the unit's energy is regained.

- FOV – Maximum radius at which the Unit can detect other objects. Used when constructing Influence Maps, and for NPC controllers.
- Primary Weapon
- Secondary Weapon – Can be null in the case of NPC units.
- Shield Type – Specifies resistance of shields to machine gun fire (from GDD):
 - Light : 100% damage
 - Heavy: 10% damage
 - Player: 0 damage
- Base Mesh – The mesh to use for the base (tread) section of tanks, or mount for stationary turrets.
- Base Texture – Texture to use on the base mesh while the unit is alive.
- Base Destroyed Texture – Texture to use on the base mesh while the unit is not alive.
- Turret Mesh – The mesh to use for the turret section of the unit.
- Turret Texture – Texture to use on the turret mesh while the unit is alive.
- Turret Destroyed Texture – Texture to use on the turret mesh while the unit is not alive.

4.9.3 Functions

- Update (float dt) – Calculates unit's new position and rotation based on current state.
- Initialize () – Sets the initial position of the unit, rotation, max health, max speed, etc.
- Damage (Weapon type) – Decreases the unit's current shields based on damage and Shield Type.
- Accelerate (int direction) – Increases or decreases the unit's forward speed.
- Rotate (int direction) – Sets the Rotation speed to the left or right.
- Aim Direction (float x, float y) – Aims the gun toward the given 2D point (heading).
- Deploy () – Causes the unit to play the deploy animation, if it exists.
- Fire Primary () – Fires the primary gun in the direction the turret is currently rotated toward
- Fire Secondary () – Fires the secondary gun in the aim direction (if exists)

4.10 Weapon

A weapon structure is responsible for describing the weapon(s) fired by a given unit. All weapon types are defined by Lua script. Weapon types are stored in the Weapon Manager, and references to them are passed to Units on construction.

- Name – Used as a key to match Units with their weapons
- Damage – How much damage the weapon does upon impact
- Energy Cost – The amount of energy required to fire the weapon
- Reload Time – How much time the weapon takes to reload
- Range – How far the weapon can fire
- Area of effect – Range of the damage once it has hit
- Projectile speed – Speed at which a bullet from the weapon travels
- Is Artillery – Artillery shells can travel over other units

4.11 Bullets

Weapons fire bullets. As there may be hundreds of bullets on screen at a time, with many of them being created and destroyed every second, the game will utilize a pool of bullets that can be reused as needed.

- Position
- Owner
- Velocity
- Damage
- TimeToLive

4.11.1 BulletManager

The bullet manager is responsible for initializing all of the bullets as well as creating and deleting them as necessary.

4.12 Controller

Controller is a base class used to represent a mechanism for controlling Units. Subclasses can be Players, NPC controllers, or Turret controllers.

Functions

- Update()

Attributes

- Team – Current team the player belongs on.
- Unit – Pointer to the unit instance which the player controls.

4.12.1 Player

A player class is responsible for many things: controlling its tank, receiving input and conveying the commands to the tank. Player will also store player pertinent information for the current session including shots fired, enemies killed, health regenerated, etc.

Attributes

- Deaths – How many times the player has died during gameplay.
- Shots fired – Counts the number of shots fired by the player.
- Enemies killed – Counts the number of enemies killed by the player.
- Damage taken – Counts the total amount of damage taken by the player.
- Health regenerated – Counts the total amount of health regenerated by the player.
- Time played – Total of time played by the player.
- Name/ID – Name/Identification.

4.12.2 NPC Controller

NPC Controllers are discussed in detail above under AI System, Low-Level Behaviors.

4.12.3 Turret Controller

Turret Controllers govern the operation of stationary turrets in the game. Stationary turrets have the simple behavior of choosing the closest enemy unit in range as a target, and firing at it until it dies, or leaves. Then a new target is chosen. Turrets use a set firing pattern, implemented as a list of time delays. A turret controller fires its weapon, waits for it to reload, and waits for the delay specified in its firing pattern, and fires again, incrementing the index in the delay list.

4.13 Destructible Object

Some objects in the world can be destroyed. These objects will have health and when their health runs out; they are removed from the world and replaced by a model of rubble. Destructible objects that register as collidables, will be removed when destroyed. Rubble objects will have a timer that dictates how much time they will be on the screen and a function that dictates how these items, fade, or disappear from the world.

4.14 Indestructible Object

Other objects in the world cannot be destroyed, such as large rocks or trees, and cannot be modified during gameplay.

4.15 Objective

The objective class defines the starting conditions and terminating conditions of an objective. The mission class manages active and inactive objectives. Objectives will be responsible for managing their timer if they have one associated with it. They will also be responsible for triggering spawning events that are associated with this event.

4.16 Mission

When the mission class gets initialized it will load an environment and objectives from a mission file. The mission class will be responsible for triggering objectives. These triggering include activating units, starting timers, or detecting state of an object.

4.16.1 Mission File Syntax

Mission format (LUA/XML):

- Header
 - Name
 - Description
 - Level File
 - Thumbnail
- Music
 - File List
- Sounds
 - File List
- Objectives
 - Name

- Description
- Object Id (trigger)
- Type (Destroy, Defend, Timed)
 - Destroy
 - Building or Tanks
 - Buildings objects Id List
 - Tanks number of objects to destroy
 -
 - Defend
 - Objects Id List
 - Timer (optional)
 - Timed
 - Object Id (used for trigger)
 - Length of time

4.17 Level

A level will be responsible for the rendering of the terrain and objects. The height map and texture information will be loaded from this file. The file definition will also contain information about what objects to load, along with where they are located on the map. The level will be responsible for loading and positioning each of these objects; the scene graph will be responsible for deciding what objects get rendered and perform these optimizations for this.

4.17.1 Level Parsing

Information about the level object, its implementation within the application, the level file format, etc.

4.17.2 Level File Requirements

The level file format needs to describe the game environment, the characters in the world, and the objectives and events specific to that level or mission

MISSION INFO:

Player Spawn Points, Objectives, Events, Enemy Spawn, Player Spawn, Enemy Start Locations (individual and groups), Friendly Spawn Locations (individual and groups), Regions, Destructibles??, Special Objects, Start Sequence. AI???

4.17.3 Level File Syntax

Level format (Binary):

- Header
 - Ident
 - Version
 - Map size in tiles

- Tile size
- Terrain vertex information
 - Heightmap
 - Normals
 - X Y
 - U V
- Index information
- # of textures
 - Texture Header
 - Size
 - Name
 - Usage?
- # of models
 - Model Header
 - Size
 - Name
 - Count
 - Location
 - Scale
 - Orientation
 - Visible
- # of Lights
 - Light information

4.18 Session

The session will be responsible for storing information gathered over the course of a level including: time passed, units spawned, etc. It will also keep track of the players and objective status.

5 Networking Protocol

The networking protocol is too large to fit in this document and will be extended to a new document as development continues. At a high level there are various protocols that will be required for the development of this game.

The first protocol is broadcasting the hosting of a game. The way the testing environment is setup it requires us to broadcast the game to open the firewall and establish connection with clients looking for a game.

The next protocol will involve a chat protocol. This protocol will be setup on a different network that does not pass game specific information unless this information is required. The chat protocol will be opened in the game lobby and stay open though playing a game.

The loading screen this protocol will need to coordinate between all clients to make sure that each client has fully loaded all the game content and is ready to start the game.

The in game protocol will be the largest protocol and handle passing packets that update unit movement, life, and state changes. This protocol will be responsible for keeping the server and all clients in sync and have the same game state so each client has the same game experience.

Finally a disconnect protocol will need to be in place if a client drops the game this protocol will alert the server and all other clients of the dropped client and decide what they can do.

6 Content

This is content. Anything that will affect the player experience if changed will go here. This includes: controls, unit statistics.

6.1 Influence Map Parameters

6.2 Commander AI Parameters (HIGH LEVEL)

In this section we talk about how the AI is configured. Do not talk about architecture; state machine vs. decision tree. Instead we talk about tweakable values that we can change to make the AI work the way we want it to and what those values are.

In the high level AI the variables that allow the system to work are ones that control group size, influence map scale, influence map view, and network inputs. The group size is the radius around a single unit that another unit needs to be contained within before they are considered as part of the same group. The influence map scale determines the ratio of the influence map to the actual map and defines the resolution for each unit's area of threat. The influence map view defines how many square units of the influence map should be used as inputs to the neural network. This defines the area around a group of units that the neural network will analyze to make its decision.

6.3 Low Level AI Parameters

6.4 Console Commands

This section specifically focuses on the console scripting system that allows for a user, or a developer to modify the game as it is running. These are some of the features it will have.

- Change window properties
 - Resolution
 - Full screen
- View / Change properties on Units (Nodes)
 - Health
 - Position
 - Ammo stats

- Weapon stats
- View Debug Information

Most everything we want to do with a console command should always be run through the Lua command. This might slow some things down, so should be tested.

6.5 User Configurable Game Options

- Volume
- Difficulty (maybe not a game setting, but something you select when you start a new game)
- Customize Controls
- Resolution
- Toggle Full Screen

6.6 User Interface Components

This needs to be broken down by state. Each state will have several components to consider.

Eg. Splash Screen... image pane ... uses image “logo.png” – a picture of our logo

6.7 Controls

This section explains the control system through various stages of the game.

The following is a list of commands that can be sent to a tank

- Forward
- Back up
- Turn Left
- Turn Right
- Fire Main Gun
- Fire Alternate Gun
- Pause the game
- Zoom camera in
- Zoom camera out

6.7.1 Tank Controls

The default mapping will be as follows:

Action	Keys
Forward – this drives the tank in the direction its facing	Up / W
Backward – this drives the tank in the opposite	Down / S

direction its body is facing	
Rotate Body Counterclockwise	Left / A
Rotate Body Clockwise	Right / D
Rotate Turret – this dictates where the turret will rotate towards	Mouse Movement*
Fire – this will fire the tank's primary weapon	Left Mouse Button (LMB)
Fire Alternate – fires the alternate weapon	Right Mouse Button (RMB)
Pause – this will bring up the pause menu	Escape
Change camera zoom – Zooms the camera in and out from a minimum and maximum distance.	Mouse wheel (z) up and down. Down on the wheel zooms out, up zooms in.

*Turret will face the direction the mouse cursor is targeted to.

These will be defined in a Lua configuration file.

6.8 NPC Classes

This section lists attributes for the NPC classes, along with their weaponry.

The *Attributes* describe the specific implementation of each of the attributes that govern the unit, except behavior. The *Primary Weapon* field lists the attributes of the unit's primary weapon.

6.8.1 Wolfhound

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	100
Max Turn Speed	<i>(seconds / revolution)</i>	2
Turret Turn Speed	<i>(seconds / revolution)</i>	1.5
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	200
Shield Regeneration Rate	<i>(numeric value / second)</i>	0
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	0%
Field of View	<i>(meters)</i>	100
Primary Weapon	<i>(string)</i>	Wolfhound_Primary
Secondary Weapon	<i>(string)</i>	(none)
Shield Type	<i>(string)</i>	Light

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Wolfhound_Primary
Damage	<i>(numeric value)</i>	25
Energy Cost	<i>(Percentage of total energy)</i>	0%
Reload Time	<i>(seconds)</i>	0.5
Range	<i>(meters)</i>	40
Area of Effect	<i>(meters)</i>	5
Projectile Speed	<i>(meters / second)</i>	500
Is Artillery	<i>(Boolean)</i>	No

6.8.2 Durant

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	120
Max Turn Speed	<i>(seconds / revolution)</i>	2
Turret Turn Speed	<i>(seconds / revolution)</i>	1
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	150
Shield Regeneration Rate	<i>(numeric value / second)</i>	0
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	0%
Field of View	<i>(meters)</i>	50
Primary Weapon	<i>(string)</i>	Durant_Primary
Secondary Weapon	<i>(string)</i>	(none)
Shield Type	<i>(string)</i>	Heavy

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Durant_Primary
Damage	<i>(numeric value)</i>	15
Energy Cost	<i>(Percentage of total energy)</i>	0%
Reload Time	<i>(seconds)</i>	0.5
Range	<i>(meters)</i>	30
Area of Effect	<i>(meters)</i>	5
Projectile Speed	<i>(meters / second)</i>	400
Is Artillery	<i>(Boolean)</i>	No

6.8.3 BRAMM

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	60
Max Turn Speed	<i>(seconds / revolution)</i>	4
Turret Turn Speed	<i>(seconds / revolution)</i>	4
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	100
Shield Regeneration Rate	<i>(numeric value / second)</i>	0
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	0%
Field of View	<i>(meters)</i>	110
Primary Weapon	<i>(string)</i>	BRAMM_Primary
Secondary Weapon	<i>(string)</i>	(none)
Shield Type	<i>(string)</i>	Light

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	BRAMM_Primary
Damage	<i>(numeric value)</i>	200
Energy Cost	<i>(Percentage of total energy)</i>	0%
Reload Time	<i>(seconds)</i>	3
Range	<i>(meters)</i>	200
Area of Effect	<i>(meters)</i>	30
Projectile Speed	<i>(meters / second)</i>	100
Is Artillery	<i>(Boolean)</i>	Yes

6.8.4 Stonewall

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	50
Max Turn Speed	<i>(seconds / revolution)</i>	4
Turret Turn Speed	<i>(seconds / revolution)</i>	3
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	500
Shield Regeneration Rate	<i>(numeric value / second)</i>	0
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	0%
Field of View	<i>(meters)</i>	75
Primary Weapon	<i>(string)</i>	Stonewall_Primary
Secondary Weapon	<i>(string)</i>	(none)
Shield Type	<i>(string)</i>	Heavy

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Stonewall_Primary
Damage	<i>(numeric value)</i>	100
Energy Cost	<i>(Percentage of total energy)</i>	0%
Reload Time	<i>(seconds)</i>	2
Range	<i>(meters)</i>	50
Area of Effect	<i>(meters)</i>	10
Projectile Speed	<i>(meters / second)</i>	500
Is Artillery	<i>(Boolean)</i>	No

6.9 Player Classes

This section lists attributes for the player classes, along with their weaponry.

The *Description* field is the summary that shows up when a player is selecting a vehicle in the Solo Game Setup and Team Game Setup rooms. It must only be a few sentences long. The *Attributes* describe the specific implementation of each of the attributes that govern the unit. The *Primary Weapon* field lists the attributes of the unit's primary weapon. The *Secondary Weapon* section lists the attributes for the machine gun common to all player units.

Secondary Weapon:

Attribute Name	Value Type	Value
Name	(string)	Player_Machinegun
Damage	(numeric value)	1
Energy Cost	(Percentage of total energy)	0%
Reload Time	(seconds)	0.05
Range	(meters)	40
Area of Effect	(meters)	0
Projectile Speed	(meters / second)	1000
Is Artillery	(Boolean)	No

6.9.1 The Hornet

Description: A description of the hornet that player's can see

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	100
Max Turn Speed	<i>(seconds / revolution)</i>	2
Turret Turn Speed	<i>(seconds / revolution)</i>	1.5
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	200
Shield Regeneration Rate	<i>(numeric value / second)</i>	50
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	50%
Field of View	<i>(meters)</i>	75
Primary Weapon	<i>(string)</i>	Hornet_Primary
Secondary Weapon	<i>(string)</i>	Player_Machinegun
Shield Type	<i>(string)</i>	Player

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Hornet_Primary
Damage	<i>(numeric value)</i>	25
Energy Cost	<i>(Percentage of total energy)</i>	5%
Reload Time	<i>(seconds)</i>	0.5
Range	<i>(meters)</i>	40
Area of Effect	<i>(meters)</i>	5
Projectile Speed	<i>(meters / second)</i>	500
Is Artillery	<i>(Boolean)</i>	No

6.9.2 The Rhino

Description: A description of the Rhino that player's can see

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	50
Max Turn Speed	<i>(seconds / revolution)</i>	4
Turret Turn Speed	<i>(seconds / revolution)</i>	3
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	500
Shield Regeneration Rate	<i>(numeric value / second)</i>	100
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	25%
Field of View	<i>(meters)</i>	75
Primary Weapon	<i>(string)</i>	Rhino_Primary
Secondary Weapon	<i>(string)</i>	Player_Machinegun
Shield Type	<i>(string)</i>	Player

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Rhino_Primary
Damage	<i>(numeric value)</i>	100
Energy Cost	<i>(Percentage of total energy)</i>	50%
Reload Time	<i>(seconds)</i>	.5
Range	<i>(meters)</i>	50
Area of Effect	<i>(meters)</i>	10
Projectile Speed	<i>(meters / second)</i>	500

Is Artillery	<i>(Boolean)</i>	No
--------------	------------------	----

6.9.3 The Lancer

Description: A description of the Lancer that player's can see

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	60
Max Turn Speed	<i>(seconds / revolution)</i>	4
Turret Turn Speed	<i>(seconds / revolution)</i>	4
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	100
Shield Regeneration Rate	<i>(numeric value / second)</i>	50
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	25%
Field of View	<i>(meters)</i>	110
Primary Weapon	<i>(string)</i>	Lancer_Primary
Secondary Weapon	<i>(string)</i>	Player_Machinegun
Shield Type	<i>(string)</i>	Player

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Lancer_Primary
Damage	<i>(numeric value)</i>	200
Energy Cost	<i>(Percentage of total energy)</i>	50%
Reload Time	<i>(seconds)</i>	2
Range	<i>(meters)</i>	200
Area of Effect	<i>(meters)</i>	30
Projectile Speed	<i>(meters / second)</i>	100

Is Artillery	<i>(Boolean)</i>	Yes
--------------	------------------	-----

6.9.4 The Ant

Description: A description of the Ant that player's can see

Attribute Name	Value Type	Value
Max Speed	<i>(meters / second)</i>	120
Max Turn Speed	<i>(seconds / full 360 degree rotation)</i>	2
Turret Turn Speed	<i>(seconds / full 360 degree rotation)</i>	1
Acceleration	<i>(meters / second²)</i>	
Max Shields	<i>(arbitrary numeric value)</i>	150
Shield Regeneration Rate	<i>(numeric value / second)</i>	50
Energy Regeneration Rate	<i>(Percentage of total energy / second)</i>	25%
Field of View	<i>(meters)</i>	75
Primary Weapon	<i>(string)</i>	Ant_Primary
Secondary Weapon	<i>(string)</i>	Player_Machinegun
Shield Type	<i>(string)</i>	Player

Primary Weapon:

Attribute Name	Value Type	Value
Name	<i>(string)</i>	Ant_Primary
Damage	<i>(numeric value)</i>	15
Energy Cost	<i>(Percentage of total energy)</i>	10%
Reload Time	<i>(seconds)</i>	.2
Range	<i>(meters)</i>	30
Area of Effect	<i>(meters)</i>	5
Projectile Speed	<i>(meters / second)</i>	400

7 External Tools / Utilities

7.1 Level Editor

The level editor that is going to be used is "Grome". This editor will have a custom exporter built for it that will allow for easy importing into our game. The details of this tool will be described as development begins as it is unknown as to how much we can export from this tool.

7.2 Maya Exporter

An exporter has been written for Maya that allows us to export static models and animated models into a binary format. This exporter may be updated as the development process progresses but things that are required for the exporter are a static mesh, Animation information, and Locator node information.

The static mesh requires vertex information as X, Y, Z coordinates, a normal for each vertex, a UV coordinate for each vertex and index information to build the model as a triangle strip. This is the minimum amount of information for this one file.

The animation information is a second file that contains joint, weights, and keyframe information to animate an object. This can be contained in the same file as the static mesh, but for simplicity of importing a static mesh over animation they are not.

7.3 Font Sheet Generator

The font sheet generator is an external tool that can be used to work with the GUI system for the game. The font sheet generator takes a TTF font file as an input and a font size and sheet size and will export a PNG of the characters laid out on the sheet and a corresponding XML file describing the character coordinates and spacing attributes for a set of characters.

The Font Sheet Generator will be developed by the team and will be written in Flash / ActionScript AS3 and will be exported as a stand-alone AIR application.

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Art Bible

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

TREND BOARDS

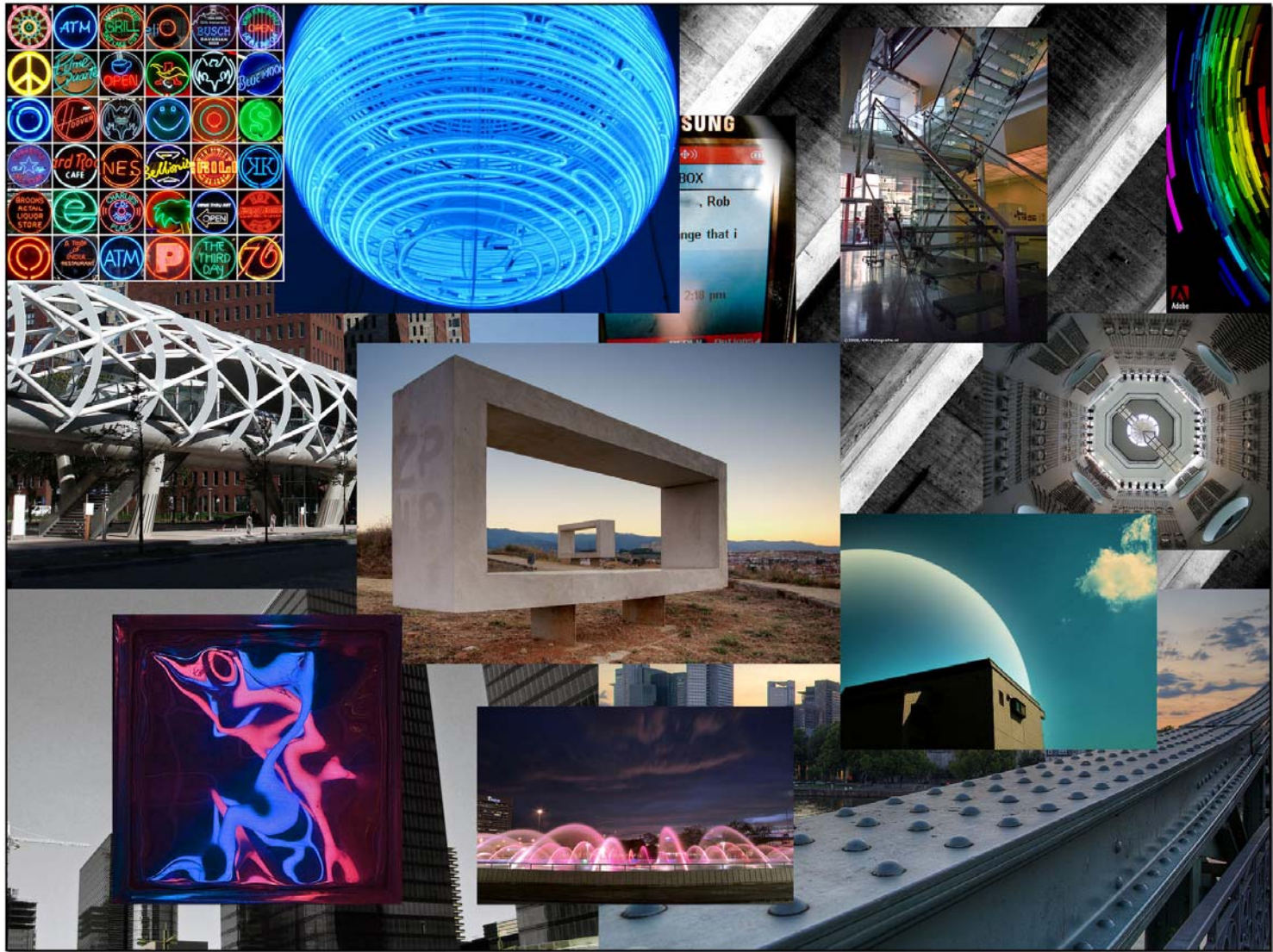
Trend boards are the mind-map of the graphical world. Trend boards are a collection or collage of several images that are put in place to inspire and help the mind escape it's envelope.

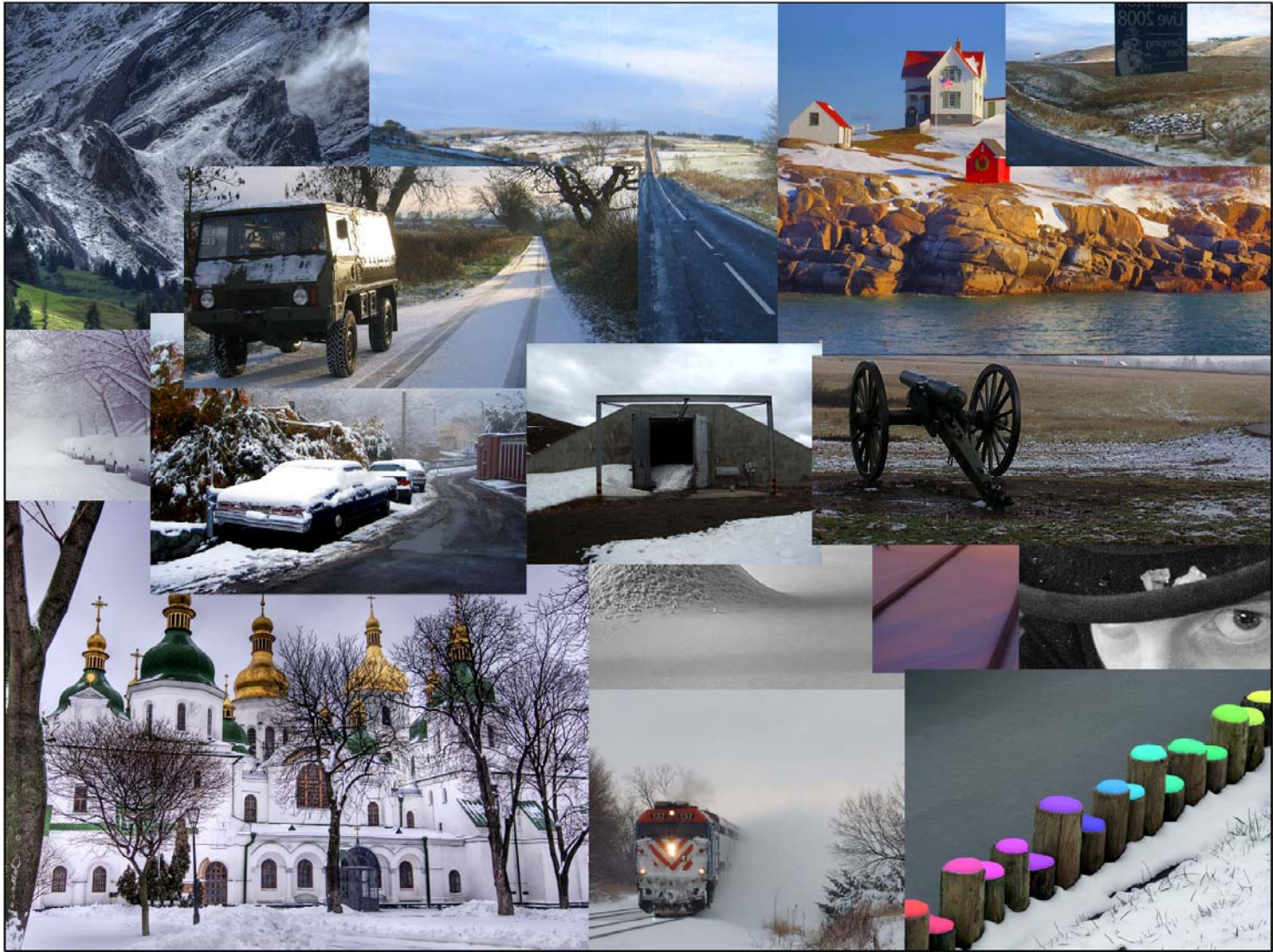
Three trend boards were compiled for the game.

The first trend board aims at the oil, grunge, and industry that the game describes. Heavy brick, red tones, fire, rust, graffiti and steel.

The second trend board targets the futuristic aspect of the game. Despite the post apocalyptic setting, neon lights, cool blues, and modern concrete designs set a theme for the future.

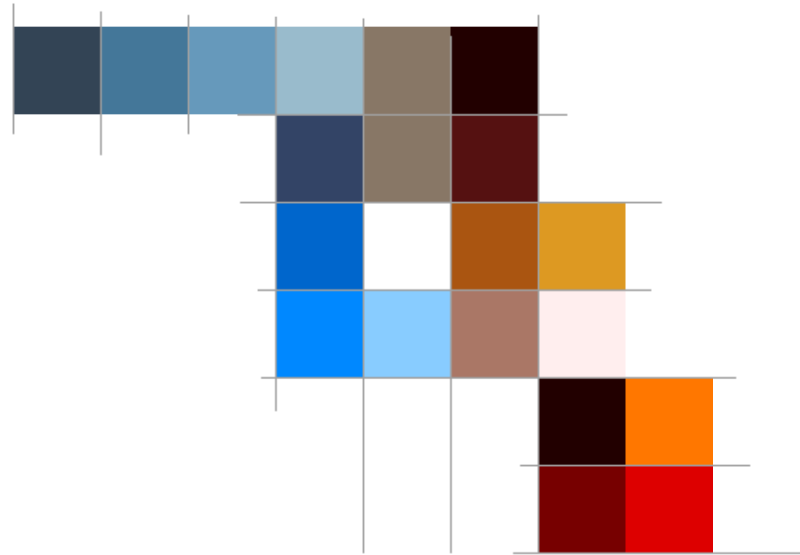
The final trend board targets the environment that the game takes place in. Snow, cool blue rivers, sunsets, and vehicles driving down the road.





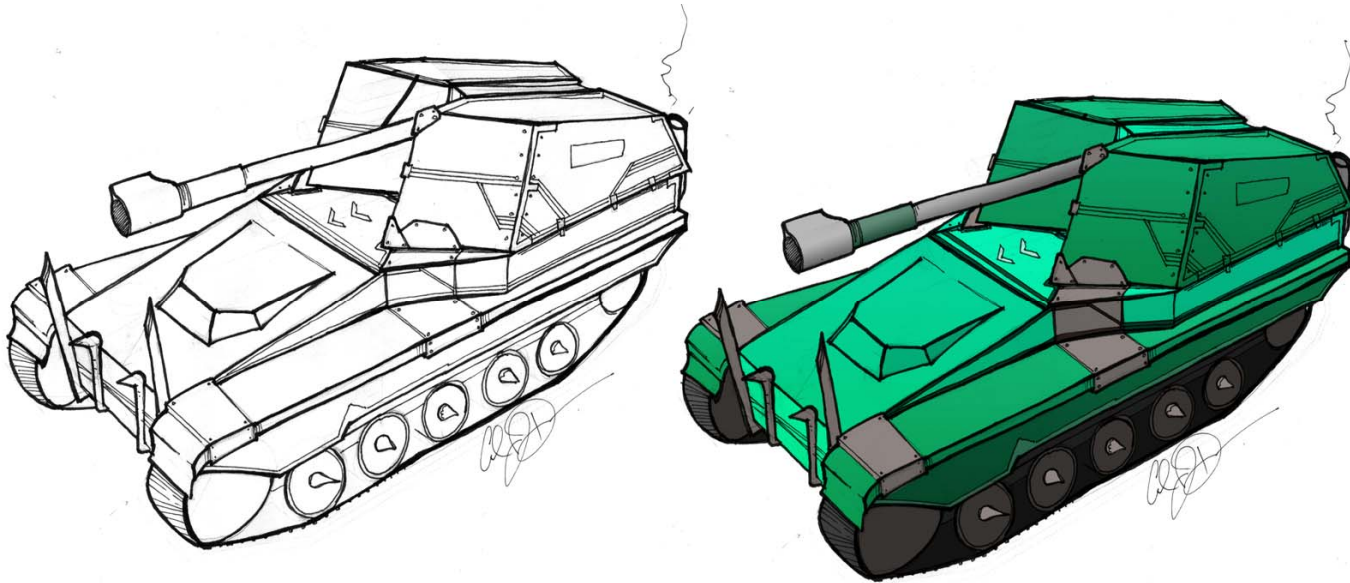
COLOR THEMES

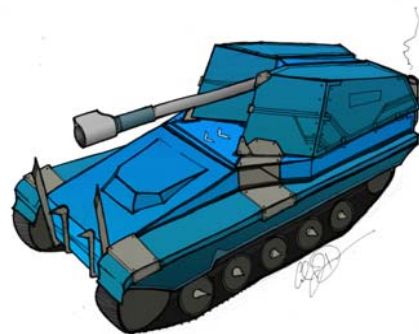
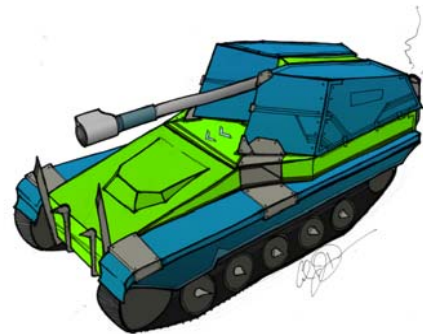
From the trend boards and similar images, we are able to derive a color palette for the game.



CONCEPTUAL ART

In order to create conceptual art, we started by referencing real world vehicles. We found vehicles that we felt fit the tank classes. Each vehicle then had a wireframe mesh generated for it. This wireframe mesh would then be sketched into conceptual 2D art.



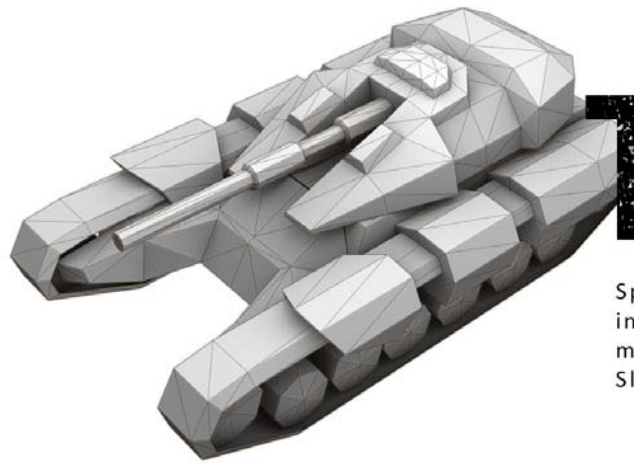


COLOR TESTS

Color tests were conducted to see which combinations of colors would work well together, and to hopefully unveil which sets of contrasting colors would work well together, should we need higher contrast for our vehicles.

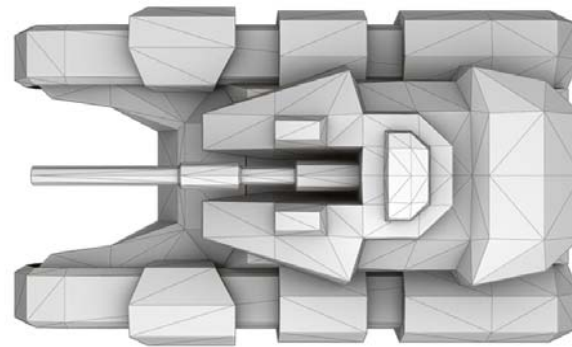
This color test was on the original lancer conceptual art.

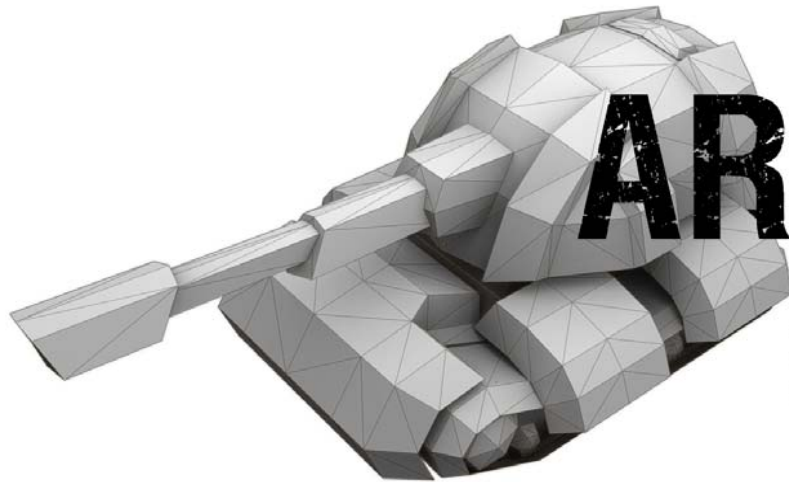
None of the color combinations proposed have been accepted as the color combinations that we will be using in our game.



THE ANT

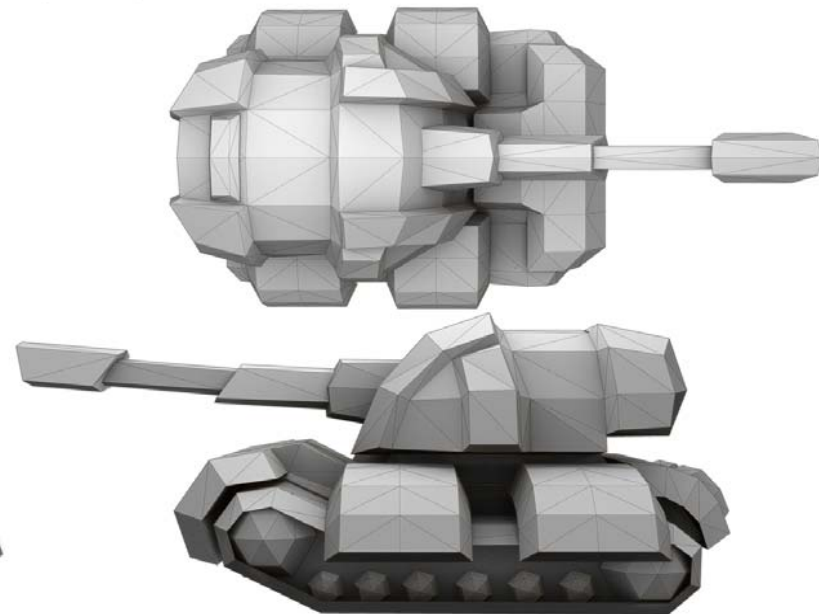
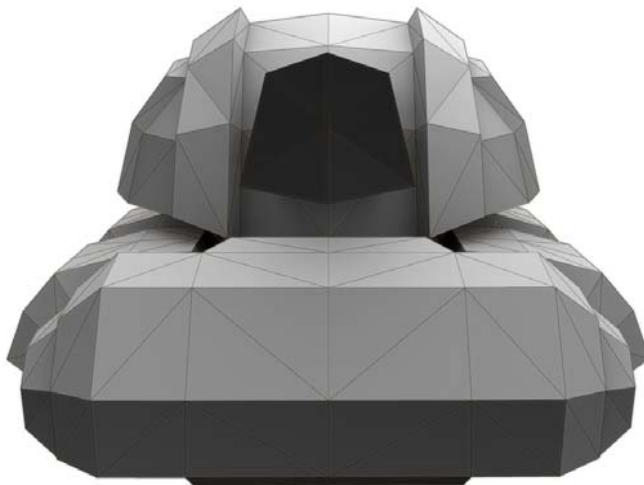
Specializing in hit and run tactics, the ant instills fear in larger, slower tanks. Inspired by the shape of modern day speedboats with its two long treads. Slender design.

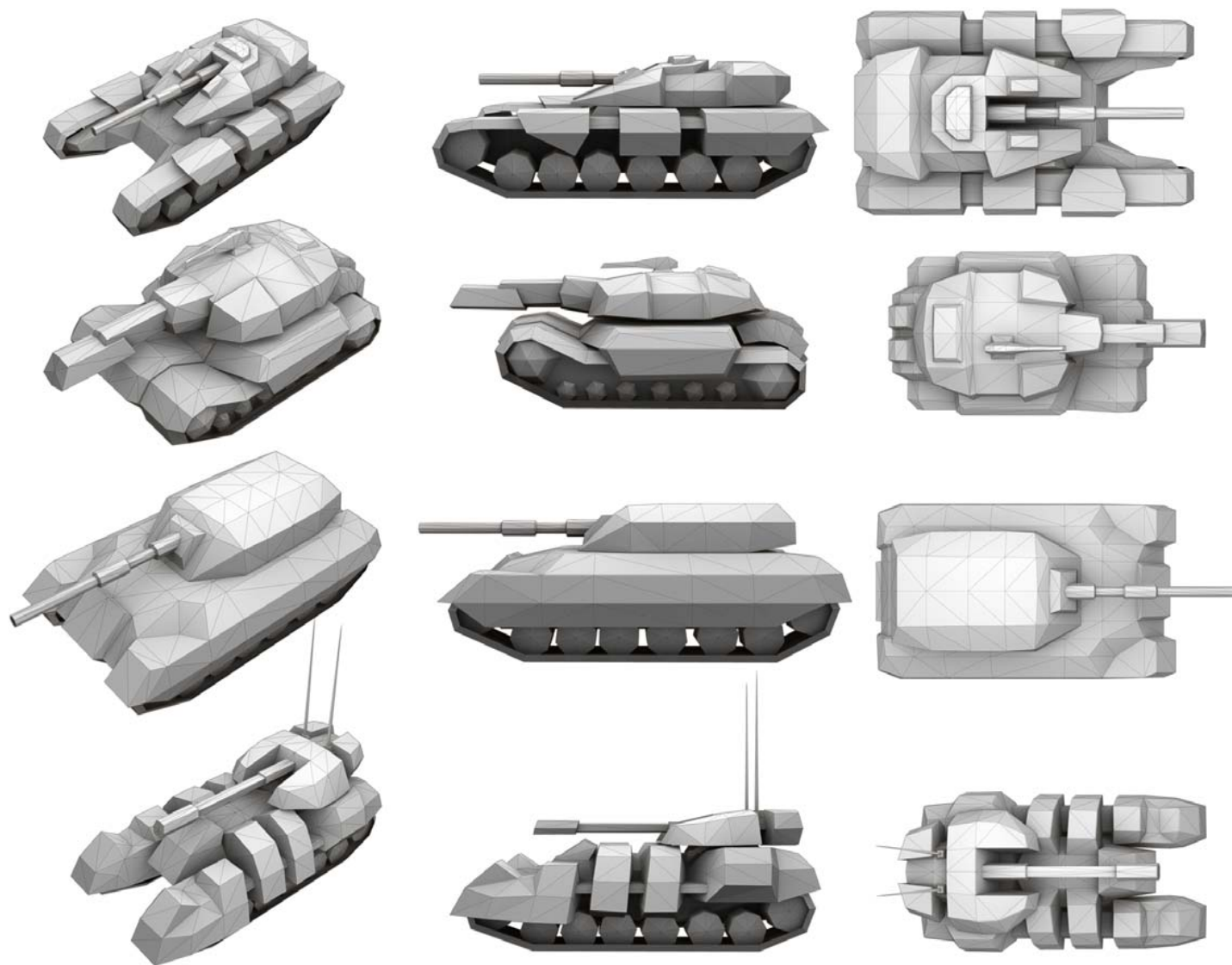




ARTILLERY

Slow to set up but quick to end a battle, the artillery is any soldier's best partner. A stout vehicle that can pack a punch.

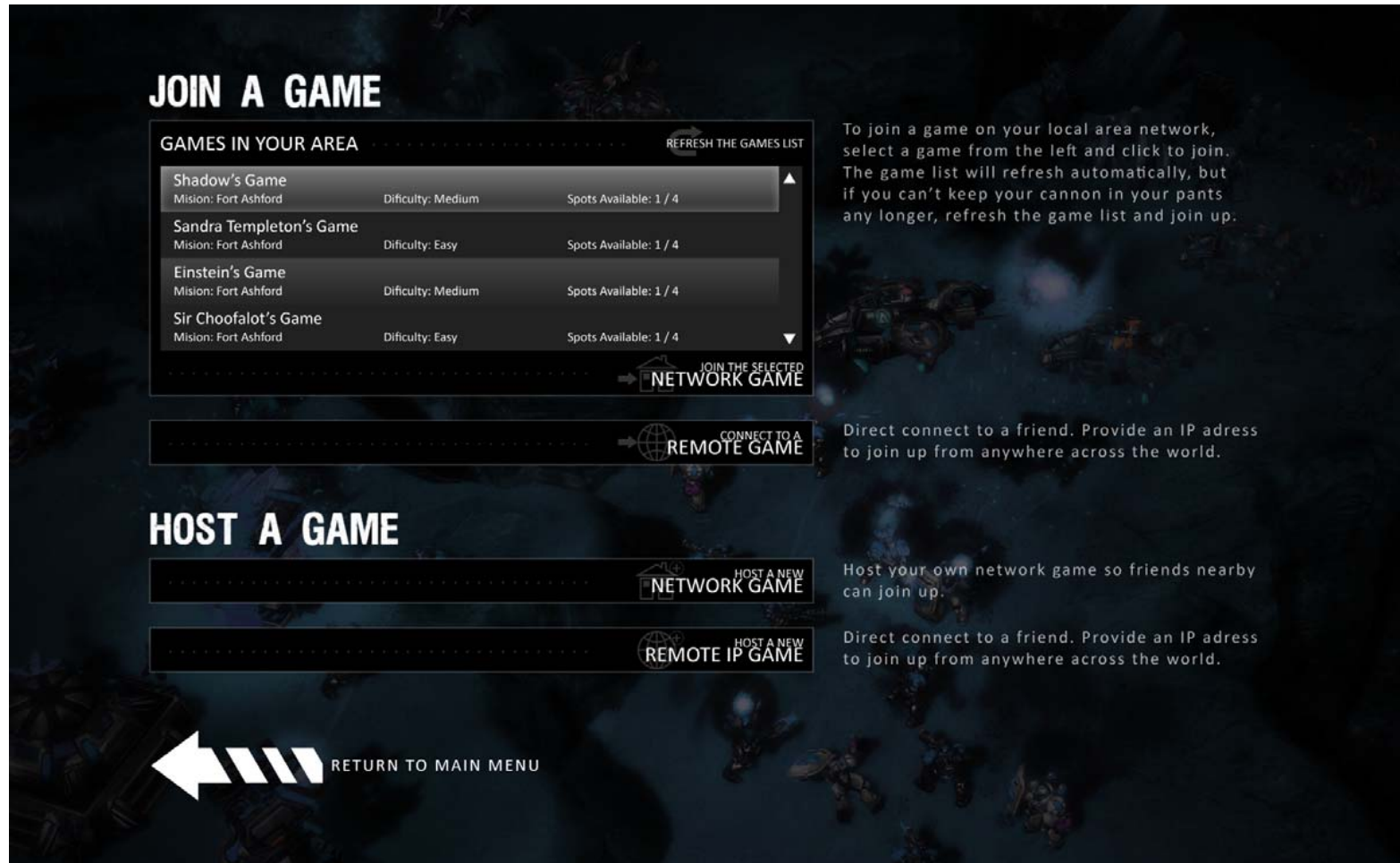




PLAYER TANK CLASSES

GAME STATE COMPOSITIONS

The Team Game Selection state. Likely the final style for the user interface systems.

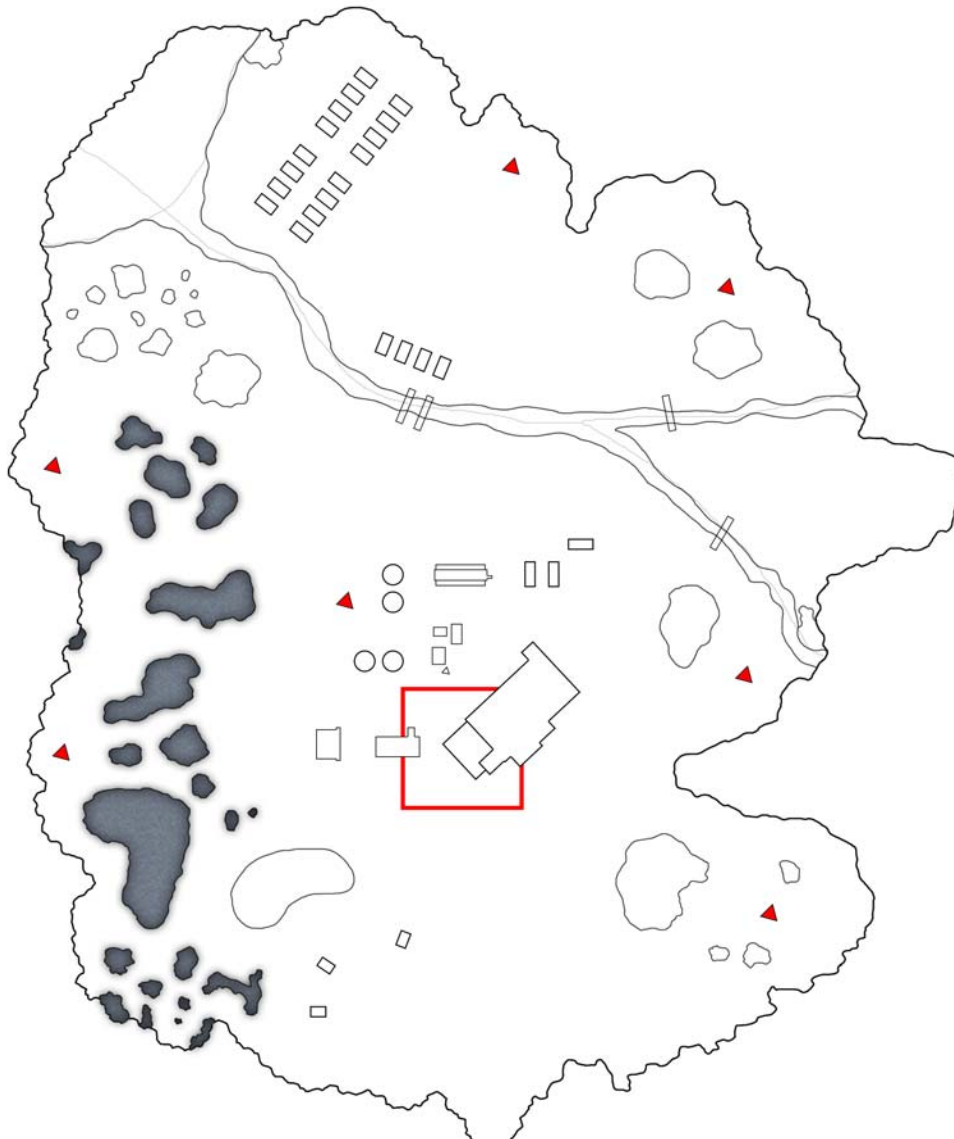


An early conceptual test of the menu system.



LEVEL 01: FORT ASHFORD

The initial level for the game



Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Individual Research Documents

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Deferred Shading

Matthew Bozarth

Rochester Institute of Technology

21 Imperial Ave

Ventura, CA 93004

<http://raydenuni.com>

bozarth@alumni.rose-hulman.edu

ABSTRACT

In this paper, the author looks at the history of deferred shading, how it has been implemented in an AAA-title, the benefits and detriments, and how the author plans to write his own.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Deferred shading, DirectX 10, game programming, lighting.

1. INTRODUCTION

What is deferred shading? Before we can really talk about deferred shading we have to talk about a few other things first: what is shading, what is a forward rendered pipeline, and then we can talk about how deferred shading differs and how it implements shading. Shading is a method of simulating lightness and darkness on a material in a 3D computer graphics scene that has lighting. In a shading model like Phong, the shading at a particular point on a surface is calculated using the normal, the direction of the light, and the position of the camera. When using a standard forward shading process, each of the inputs is determined for each pixel of each surface on the screen. That is, the complete shading process is run for every pixel before the next pixel is rendered. Deferred shading does not do this. It instead calculates the normals for each pixel and renders that to a texture, renders the depth of every pixel to a texture, and renders the color, without lighting, to a texture. It then uses these three textures as input to a second pass that is able to calculate the shading for every pixel at once, reducing the need to possibly render shading for a surface that is occluded, and in some ways, simplifying the shading pipeline. There are benefits reaching beyond a simplified pipeline and screen space shading as well. Because the shading is done in screen space, more advanced lighting calculations can be done in linear time. This method is being looked at for a rendering engine I will be writing for an upcoming game.

2. HISTORY

Michael Deering proposed the idea in a 1988 paper at SIGGRAPH, although he never used the term “deferred shading.”[8] Decades later, the game industry realized the potential for this approach in real time graphics and people began to take notice. It wasn’t until 2004 with the talk given at GDC

entitled “Deferred Shading and Lighting”[13] and the “6800 Leagues Under the Sea: Deferred Shading” presentation given by Nvidia that the ball really started rolling. Within a year, the developers behind S.T.A.L.K.E.R. released a paper, which can be found in GPU Gems 2, on their experience building an engine solely based on deferred shading. Three years later, in 2008, GPU Gems 3 came out with a follow up article on deferred shading written by the developers of Tabula Rasa on their experience building a deferred shading based engine. “Deferred Rendering in Killzone 2” was presented in 2007 and at SIGGRAPH 2008, Blizzard presented a paper on the effects and techniques they used in Starcraft 2 which included a section on deferred shading. Many AAA titles are being developed with a deferred pipeline in mind, and the trend doesn’t seem to be letting up. In contrast to all the AAA development, Riemer Grootjans included an implementation tutorial of the technique in his book, “XNA 2.0 Game Programming Recipes: A Problem-Solution Approach”[4] bringing deferred shading to the masses.

3. EXAMPLES

At this point, a few key papers and presentations stand out as the authoritative pieces on the subject, including: “6800 Leagues Deferred Shading” and “Deferred Shading in S.T.A.L.K.E.R.” I will be briefly summarizing and discussing the techniques presented and implemented. For more on implementation, I recommend the paper on Tabula Rasa[3], STarcraft II[7], the Leadwerks Engine[9], and the Killzone 2[14] presentation.

3.1 6800 Leagues: Deferred Shading

As one of the initial talks on deferred shading, “6800 Leagues Deferred Shading” is the source every other article references when introducing the topic. So what does it say? It wastes no time declaring that one of the major purposes for this technique is lighting. Modern games are pushing the number of lights and forward shading really isn’t set up to handle this level of scaling. The issue with a forward shader and increasing the number of lights is that the complexity of the lighting increases exponentially with the complexity of the scene. In a deferred shader, because lighting happens in screen space, the lighting complexity is disconnected from the scene complexity.

The presentation mentions the typical issues everyone implementing a deferred renderer has to deal with: all render targets must have the same bit depth which leads to wasted space, point lights are hard, there’s no transparency, and hardware AA does not work. It’s a little unsatisfying to see that some of the initial work on this area discovered these problems and four years later no one’s been able to solve them. Some problems just don’t have elegant solutions. Despite having all the basic concepts written down, the authors were unsure about whether deferred shading was going to take off or not. It just hadn’t been around

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

long enough and not enough people had implemented it to know if it was worth it.

3.2 S.T.A.L.K.E.R.

The team behind this game spent two years researching and developing their engine. What makes it stand out is their renderer is “based solely on deferred shading and 100 percent dynamic lighting, targeted at high-end GPUs.”[2] Their paper is a post mortem on their research and implementation, focusing on what they learned, how did they implemented a deferred renderer, and how they dealt with the problems that arose.

The author bases his reason for choosing a deferred renderer on games still being CPU limited and a deferred renderer reduces CPU usage in exchange for a higher load on the GPU. This will of course vary from game to game, and is something to keep in mind when deciding whether to use deferred shading or not. In their case however, deferred shading does seem to have worked well. They offer screenshots of their forward rendered engine compared with their deferred rendered engine, and although deferred rendering offers no inherent graphical improvements, it does allow for more advanced techniques at a lower cost.

3.2.1 Issues

One of the first things addressed are myths concerning deferred shading including: it is slower on current hardware, it is useless for directional lights, and it is incompatible with a traditional material system. The first one is interesting because it focuses on the GPU load for a deferred renderer vs. a forward one. In a traditional forward renderer implemented on a fixed-function pipeline, it is important to balance the load on the vertex shader with the pixel shader and a deferred shader moves much of its computation off the vertex shader and onto the pixel shader, clearly not ideal. However, with the advent of a unified architecture, like that in DirectX 10, this is nearly irrelevant as the GPU will automatically load balance. For the issue of directional lighting, this is an issue for a scene with “few un-shadowed directional lights; but it’s false for even a single, shadow-casting directional light (such as the sun).”[2]

Concerning materials, it is true that a deferred renderer may be incompatible with a traditional material system; however it offers alternatives that can not only be fast, but are much more elegant. While they don’t go into much detail in this paper, after looking at other papers, such as Zima’s article “Deferred Shading in XNA,” [12] that talk about stored lighting models, it looks rather elegant. By storing the output of the lighting model in a texture and indexing into the U, V coordinates with the N.L and N.H values as well as a material ID, one is able to look up lighting calculations very quickly without having to do any lighting computation on the spot. This works nicely for a deferred shader because it associates the lighting model with the material as opposed to with the light. By putting the lighting model in a texture, it can simply be included with the material. This method also gives more control to the artists, which is always a benefit.

3.2.2 Optimizations

The goal of the team was to run at 30fps on a Geforce 6800 at 1024x768 with settings at maximum. They provided these numbers for their game:

- Avg # lights per frame in closed spaces: 50
- Avg # of lights per pixel in closed spaces: 5
- Avg # of lights per pixel in open spaces: less than 1

By putting such an emphasis on shadows, this really forced itself as a bottleneck and something that needed to be optimized. The author lists their optimization goals as:

- Limit lights drawn to those that affected the image
- Avoid rendering shadows on as many lights as possible
- Simplify lighting shaders

An interesting aspect of optimization is that when they optimized, they decided to value quality over speed for the post-processing affects because the cost of these was consistent from scene to scene whereas something like number of lights was an unknown as a player could drop torches and create as many lights as they wanted. They preferred to optimize for the unexpected rather than the expected cost.

One thing they did that I would like to look into is use the DirectX occlusion query to eliminate occluded lights. Using this query system might offer some significant performance gains for me as they found it resulted in at least a twofold increase.

To optimize lights even further, they broke down their lights based on several attributes including:

- Does the light need to cast shadows
- Does it need to project an image
- Does it cast shadows from a translucent surface
- Does the light contribute to glossy specular reflection
- Does the light move

Some of these are relatively straightforward in how they can be optimized; non-moving lights can have their shadows pre-computed; while others are a little more interesting. One reason for differentiating between lights that cast shadows and those that don’t and ones that contribute to specular and those that don’t is they used a lot of lights to simulate global illumination. Lights used for these purposes were rendered with a very simple diffuse model without shadows or a specular component.

3.3 POSITIVES

After looking at an implementation, what are the benefits of this technique? Some of them are obvious: a more streamlined, simplified pipeline, “perfect” O(1) depth complexity for lighting [1] effectively making the rendering of many small lights cost the same as a single large light. This is because it is the number of pixels covered by the light that determines the cost, not the number of polygons; adding more lights simply means a slightly longer loop. It also easily integrates with shadowing techniques. However, there are some added benefits such as allowing for an alternative, possibly more natural, material system and lighting models. Because it is a deferred approach and everything happens in screen space, any sort of screen space work is very easy to do here: bloom, HDR, even SSAO.[11]

3.4 NEGATIVES

All this sounds wonderful, but unfortunately there are some pretty serious downsides. Concerning lighting, if a game has one big directional light, this is not going to be very fast method, but that’s a relatively minor problem compared to the disappointment of not being able to use hardware accelerated anti-aliasing, although there are respectable workarounds involving screen space edge detection, made possible by the G-Buffer information, and blurring. The real killer is the complete inability to do transparencies. Because the first pass reduces scene complexity to a single depth value at each pixel, all occluded information, such

as surfaces behind a transparent surface, is thrown away. It is this behavior that makes deferred shading so excellent for lighting, but also abysmal for any materials with alpha values. Unlike the AA problem, there is no decent workaround, the only way anyone's been able to get transparency in their deferred renderer is to do a separate forward render pass. The engine for Tabula Rasa maintains two renderers, a forward and a deferred one. They use the forward renderer to maintain support for older hardware, but they also use it to render transparency as part of their deferred renderer. It works, but it isn't elegant and it requires the developers to maintain two renderers.

3.5 WHAT THEN

Is it useful then? As in almost all things, it depends on what is needed. For a game with lots of dynamic shadow casting lights, this is an excellent method that will produce a predictable performance increase. Because the method happens in screen space, increasing light complexity is non exponential. However, if there is no need for many dynamic lights; then the hassle of having to write a custom anti-aliasing method, hack transparencies, and possibly rewrite an existing material framework is probably not worth it. This method is also bandwidth intensive with all of those render-targets getting passed back and forth between the CPU and the GPU and while this isn't much of an issue on modern hardware, older hardware might have problems with it. This technique doesn't scale well with older systems, thus the Tabula Rasa engine saving their old forward renderer.

3.6 IN OUR CASE

The renderer I will be creating is for a top down game that will be rendering a terrain and a lot of tanks, much like a typical 3D RTS (real-time strategy) engine. All of the action happens outside, there's that problematic large directional light, which would seem to indicate that forward shading might work better here. One of the goals for this game however, is to do as much dynamic lighting and shadowing as possible. With possibly hundreds of particle sprites for the bullets flying around the screen, the possibility of using every one of those as a light source is very tempting and something not possible with forward shading.

Luckily, the game doesn't call for any transparency in the 3D scene, so it might be possible to get away without addressing that issue. Transparency will of course exist in the GUI and the particle system, but those will have their own render pipeline and can be added on top of the 3D scene without interfering with the deferred pass. Anti-aliasing will still be a problem, and addressing that will be a high priority. And then there are a lot of other screen-space effects that become possible, and easier, with this method: bloom, HDR, SSAO, and motion blur are all techniques that are not necessary but optional, and because they will be easier to implement, the graphical quality of the game can be improved at a lower cost.

One rather interesting side-effect from all this has to do with the material system. One of the major concerns when implementing a deferred renderer is that it will be incompatible with a team's current material framework. For this game, the plan was to build the material system from scratch anyways. Thus, there will be few issues embracing a deferred shading compatible material system, like the one used in S.T.A.L.K.E.R. and described in the Nvidia paper, "Anisotropic Lighting using HLSL." [10] By moving the lighting model from a computationally expensive algorithm to a pre-baked texture, I predict faster lighting, which is an important

optimization for a deferred shader and a more flexible system for the artists.

3.7 CONCLUSION

The goals for this project are: to implement a deferred shader that runs on DirectX 10 hardware in a way that benefits the game being developed, integrate post-processing techniques to address limitations in the technique as well as offer increased visual effects, and create a system that is easily used by artists. This won't be the first deferred renderer ever written, nor will it be one of the fanciest, but it will be my first and one of the first written in DirectX 10, which promises some benefits over the DirectX 9 that other renderers have been written in.

4. ACKNOWLEDGMENTS

I want to thank Nick Korn for lending me his copies of GPU Gems 2 and GPU Gems 3, #XNA on EFNet for discussing this with me, and Professor Andy Phelps for helping me with this topic and leading my capstone project.

5. REFERENCES

- [1] Hargreaves, Shawn, and Mark Harris. 2004. "6800 Leagues Under the Sea: Deferred Shading." Available online at http://developer.nvidia.com/objects/6800_lages_deferred_shading.html.
- [2] Shishkovtsov, Oles. 2005. "Deferred Shading in S.T.A.L.K.E.R." In GPU Gems 2, edited by Matt Pharr, pp.143-166. Addison-Wesley. Available online at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html.
- [3] Koonce, Rusty. 2008. "Deferred Shading in Tabula Rasa." In GPU Gems 3, edited by Hubert Nguyen, pp529-457. Addison-Wesley.
- [4] Grootjans, Riemer. 2008. "XNA 2.0: Game Programming Recipes." Apress.
- [5] Doolwind. 2008 "New Rendering Technique (Light Indexed Deferred Lighting)." Available online at: <http://www.doolwind.com/blog/?p=90>.
- [6] Calver, Dean. 2003. "Photo-realistic Deferred Lighting." Available online at <http://www.beyond3d.com/content/articles/19/>.
- [7] Fillion, Dominic, and Rob McNaughton. 2008. "Starcraft II: Effects & Techniques." Advances in Real-Time Rendering in 3D Graphics and Game Course – SIGRAPH 2008, Editor N. Tatarchuk.
- [8] Wikipedia contributors. 2009. "Deferred Shading." Wikipedia, the Free Encyclopedia. Available online at http://en.wikipedia.org/w/index.php?title=Deferred_shading&oldid=273038641.
- [9] Klint, Josh. 2008. "Deferred Rendering in Leadwerks Engine." Leadwerks Corporation. Available online at http://www.leadwerks.com/files/Deferred_Rendering_in_Leadwerks_Engine.pdf.
- [10] Dudash, Bryan. 2004. "Technical Report: Anisotropic Lighting using HLSL." Nvidia Corporation. Available online at http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_Aniso/docs/HLSL_Aniso.pdf.

- [11] Mittring, Martin. 2007. "Finding Next Gen – CryEngine 2." Advanced Real-Time Rendering in 3D Graphics and Game Course – SIGRAPP 2007.
- [12] Zima, Catalin. 2007. "Deferred Shading in XNA: Implementing a deferred renderer in XNA GS 2.0." Ziggyware.com. Available online at http://www.ziggyware.com/readarticle.php?article_id=155.
- [13] Geldreich, Rich, Matt Pritchard, and John Brooks. 2004. "Deferred Lighting and Shading." Presentation at Game Developers Conference 2004. Available online at http://www.gdconf.com/archives/2004/pritchard_matt.ppt.
- [14] Valient, Michael. 2007. "Deferred Rendering in Killzone 2." Develop Conference 2007, 25th July, 2007, Brighton. Available online at <http://www.dimension3.sk/mambo/Articles/Deferred-Rendering-In-Killzone/View-category.php>.

A Generic User Interface System for Games

Colin Doody
Rochester Institute of Technology
2 Fletcher Rd
Lynnfield, MA 01940
1-781-334-5649
ColinJD@gmail.com

1. OVERVIEW

Graphical user interfaces in video games are increasingly complex systems that demand increasing amounts of interaction, graphical capabilities, and support of various display modes and devices [1] [2]. Due to their dependence on the underlying rendering pipeline, the systems are typically rewritten from one engine to the next. Most generic solutions currently available are also limited in their functionality, allowing for the customization of elements within the prescribed layout and system [3]. For each game engine that is written, a new graphical user interface system is typically developed. Every game has a unique art style, and creating a generic framework for graphical user interface systems that will support these various styles is difficult.

Games that attempt to write their own graphical user interface systems from scratch may not implement or be aware of several features that could improve their user experience, either graphically or through interaction. A game, for example, may choose not to implement a comprehensive tweening system, or may not consider handling the changing of resolutions, due to the complexity or added efforts of implementing such a system. In the end, these systems may miss out on several opportunities to improve their user experience.

This research aims to propose a framework for a graphical user interface system that is generic enough to serve as many artistic visions as possible for a game. It aims to be a library that can incorporate itself into any render pipeline and accommodate the needs of any game. It aims to implement features that other graphical user interface systems have. It aims to place some responsibility in the hands of the developer to implement their vision, in turn giving it much broader application.

By exploring user interface solutions that are not typically geared towards game development, and by reviewing several game related GUI systems, this research may prove important in surfacing techniques to user interfaces that are not frequently considered for games and in potentially creating a pipeline and fundamental principles that can benefit any game project involving a user interface.

2. COMPARITIVE ANALYSIS

The following systems were studied for their user interface systems:

- Panda3D
- Adobe Flash
- Web Browsers (Mozilla Firefox)
- CEGUI

The systems yielded important information on how user interface systems are currently managed, and also on features that could benefit game systems that exist in other user interface systems.

3. SUMMARY OF FEATURES

The following list summarizes the features that will be presented in this document. These are the features of the proposed GUI system:

- Individual scenes that are able to render to a texture or render directly to a device.
- The ability to specify the output resolution of a scene independent of the scene size.
- Control of the coordinate system of the stage, independent of the output device. Enables multiple scaling features and aspect ratio adjustments.
- Receive input and handle input that is the responsibility of the GUI system, and communicate back to the application through an observer pattern.
- Text support; text highlighting and several levels of formatting control.
- Pre-defined and customizable tweening algorithms to control elements of the stage.
- Resource management that enables texture optimization through optional texture sheets and texture sets.
- Templating systems for external file definitions of scene layouts and resources.

4. SYSTEM OVERVIEW

The graphical user interface architecture attempts to internally handle as much functionality as possible without requiring direct control from the host application. The GUI system, as a whole, is responsible for:

- Rendering and Element Layout
- Resource Management
- Providing GUI specific animations and functions
- Handling GUI specific input and interaction

In order to establish a foundation of functionality as a GUI system without making too many assumptions about the application being developed, the GUI system is not responsible for:

- Creating specific widgets and windows
- Handling unrelated systems (device input management, audio)

Furthermore, for the purpose of this project, the following assumptions will be made in its implementation:

- The system will be described as a DirectX 10 application only.

5. SCENE MANAGEMENT

One aspect of the GUI system is handling elements within a scene. The user interface system makes very little assumptions

about the types of graphics and layout of objects on the screen, but offers tools to help in the presentation and interaction of different elements.

A scene in the GUI system is a hierarchy of elements, positioned in space with several properties that define their appearance and interaction. One GUI system can have several scenes, and each scene is able to render to its own target. Each scene is able to render to its own target for several reasons:

1. **Post-processing effects:** A scene that is rendered to one target can have post processing effects applied intentionally to it that other elements should not have. An example would be a system where a game has a heads-up display and when the game is paused, the game world and the heads up display fade away when a pause menu appears. If the heads-up display elements in one scene and the pause menu is in another, the heads-up display can be post-processed while the menu screen is not.
2. **Save on updating and rendering:** A single scene with several GUI elements on it may not update every frame, and so propagating the hierarchy of elements and rendering the elements on each frame may not be necessary. However, if there was an element that did update frequently, such as a frame counter, its update process should not require the other elements to update and render every frame as well. The frame counter could be placed on its own scene, updated and rendered each frame, while the more complicated scene would only need to update and render when changes occur.
3. **Creation of scenes that exist in 2D space vs. 3D space:** With the ability to create and render to multiple scenes, the ability to texture world objects with GUI scenes becomes possible. A single scene could exist that handles 2D screen elements and interaction, while a secondary scene could be set up to, for example, decorate an interactive computer display in 3D space.

5.1 Scene Elements

5.1.1 Panel

The GUI panel is the most basic scene element. A GUI panel is a single quad with a texture on it. All other elements either extend the panel or are a composite of panels.

Panels are defined by the following properties:

- **Position:** Where the origin of the panel lies, in screen space
- **Size:** The pixel width and height of the panel. This is described by the texture assigned to it, and cannot be set by the user.
- **Depth:** At what level the panel renders. Depth will allow panels to render on top of or behind other panels.
- **Color:** The RGBA color value of the panel. The color value can tint a panel a certain color, or, by using the alpha channel, can fade panels in and out.
- **Scale:** The X and Y scale of the panel. This property allows developers to control the size of the panel after its original size is set by the texture.
- **Alignment:** Alignment determines where the panels origin anchors relative to its parent. A center alignment on the stage, for example, keeps the panel in the center

of the screen. A left alignment will always keep the panel to the left of the screen. Alignment has both vertical and horizontal properties, and they can be Top, Center, Bottom, Left, or Right.

- **Origin:** Similar to the alignment, the origin specifies where, when anchored, the panel's "center" is. A panel anchored to the left with an origin on the left will remain fully on the screen, while a panel anchored left with an origin at the center will be half on the screen.
- **Rotation:** Rotation of the panel, in radians.
- **Texture:** A panel can be textured using the GUI systems internal resource system or with a provided texture resource. A provided texture resource could allow for a real time 3D render to appear on a GUI element.
- **Visible:** Whether or not the panel renders.

All panels support the parenting of other elements in the GUI scene.

5.1.2 Animated Panel

An animated panel is similar to a panel, except that it will update its texture over time. The updating of the texture over time is a function that the GUI system handles internally. When the GUI core is updated, if the scene is not flagged as paused, the Scene will get updated with the delta time that it was provided by the application. Animated panels receive this change in time, and using the time variable are able to animate themselves based on their description.

An animated panel is intended to primarily represent an animation. The animated panel can also be used to easily represent an element that changes its appearance based on a state. A checkbox, for example, is a prime candidate for an animated panel. One frame of the animated panel may reference a texture of a checked box, while the second frame will represent an unchecked box. Animated panels are easily controlled to not play by default and can toggle or switch the frame that it is on.

Animated panels reference a single sprite sheet, but also have a concept of frames. A frame in an animated panel refers to a cell in a sprite sheet and has an optional time variable. The time variable refers to how long the animated panel should wait on that frame, in seconds, before moving to the next frame.

Animated panels offer developers interactive control over content and provide basic functions for elements on the screen that change graphically. They are described in seconds in order to separate the functionality of the system away from the running frame rate of the application.

Animated panels have several functions that allow for full control over the playback of the animation. Several of the functions that describe an animated panel are derived from the MovieClip functions in the Flash ActionScript 3 library [4].

- **GotoAndPlay:** The GotoAndPlay function will accept either an integer reference to a frame index or a string name to a frame label. Calling this function on an animated panel will make the animation immediately jump to that location and will begin playing.
- **GotoAndStop:** The GotoAndStop function moves an animated panels frame to the described frame index or frame label and stops and resets all animations at that time. This function is most appropriate for buttons or

check boxes that do not animate, but instead “switch states” as described before.

- **Pause:** The pause function will stop an animation that is playing, but will maintain the time offset that has already accumulated in the animated panel.
- **Stop:** Stop will stop the animation on the frame that it is currently on and erase any accumulated time information.

Animated panels also have ways of describing looping attributes. An animated panel may chose to not loop, loop indefinitely, or loop n number of times. For this, animated panels implement the following function:

- **SetLoopCount:** The set loop count adds more control over the playback of animated panels, enabling looping events, such as may be found in a loading icon, or non looping events, which may be appropriate for an animation that only animates on its introduction.

Animated panels also have functions for controlling the speed of playback and describing the animation that should occur:

- **AddFrame:** Specifies the graphic that should be used at a frame and optionally a time for the frame to remain on screen for before progressing to the next frame.
- **SetDefaultFramerate:** Specifies the default frame rate of a frame, should one not be provided when the AddFrame function is called.
- **SetSpeedModifier:** Gives more control over the playback speed of an animated panel. Modifies the playback speed of the entire sequence.

5.1.3 TextField

Text fields share the same properties of panels, but are themselves a composite system of panels.

In typography, text is made up of characters, which are the conceptual character that is to be displayed, and glyphs, which are the actual representation of the character. A font is a collection of glyphs [5]. In the system, each glyph is represented with a Panel.

As a resource, a font is similar to a sprite sheet, which is covered later in this document, except that it contains extra functions to define how letters should be spaced next to each other.

The panel resource itself will contain the bounding information of each glyph. Additionally, the left side bearing and right side bearing, which defines the spacing that exists next to a character, either in positive or negative space, will be used to cumulatively space characters next to one another.

With characters represented as panels, and with each character having its own spacing properties, one is able to apply further typographic properties to a text field:

- **Line spacing:** The additional spacing between rows of glyphs. This value can be positive or negative.
- **Letter spacing:** The additional spacing between glyphs. This value can be positive or negative.

Direct control over glyphs and their properties enables the game engine to support the type of typographic functionality that Adobe uses in the ActionScript 3.0 TextFormatter class [6]

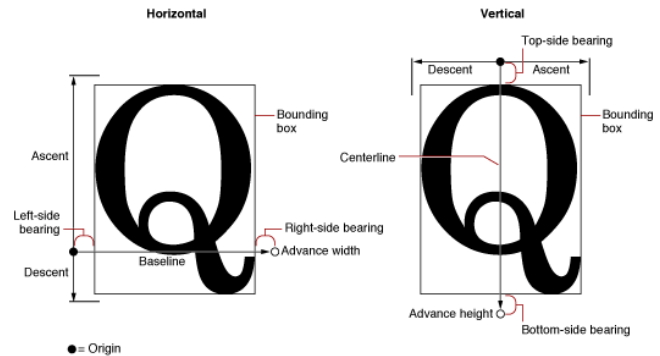


Figure 1. A diagram of the components of a single glyph. Image from Apple. [5]

5.2 Scene Layout

The layout of elements in a scene is one of the focuses of the GUI system. The layout of objects in the scene is done in different ways across different game engines and changes in the layout of the scene are similarly handled uniquely according to the specifications and demands of the game they are being implemented in.

This GUI architecture proposes a solution for scene layout that accommodates several needs of a game developer. A review of game engines and games themselves has revealed a number of techniques to positioning elements on a screen:

1. The coordinate system that GUI elements are placed on ranges from 0 to 1. This is the case in the Panda3D engine [7].
2. The coordinate system that GUI elements are placed in is in pixel space. The resolution of the output is the coordinate space that the GUI elements should be positioned in.

Questions must be asked of both means of element layout. If the coordinate system ranges from 0 to 1, how does the system handle aspect ratio changes? Does the coordinate space remain 0 to 1? And if so, how do elements remain on screen? Do elements end up stretching at this point? On the other hand, if the coordinate system is strictly in screen space, once again, how is resolution switching handled? Do the elements scale to the output resolution or do they stay in place? Do elements need to be repositioned based on the ratio of the stage, or the resolution of the output target?

This GUI system offers a simple solution that can meet the needs of any developer. It separates the concept of output resolution from stage coordinate space, and also gives the developers the freedom to specify easily their coordinate space choices and position of elements on the screen.

5.2.1 Resizing the Scene

Every stage has a stage size associated with it. The stage size is a width and height value that describes, in arbitrary units, what the coordinate space of the stage should be. When elements are positioned and sized on the screen, they will use this coordinate space.

If a developer may chose to design a system at its maximum resolution, which for some games, for example, may be 1920 x 1200. If a developer sets a stage size to 1920 x 1200 pixels, they

will need to keep in mind that a full screen background image will need to be 1920 x 1200 pixels in size, and that any single element will need to move 1920 pixels across the x axis to move from one side of the screen to another. A designer now has control over the elements on their screen at a pixel level. They can directly move an object around at a pixel level and be guaranteed its accuracy.

Despite the size of a stage, a scene in the GUI system can output to any resolution target. While it is the responsibility of the developer to ensure that the aspect ratios match, a developer can easily establish their stage at 1920 x 1200 and configure all motion of elements within the GUI to that resolution and output at 1280 x 800, maintaining the relative motion, position and size of elements without any added cost. The output procedure will also not suffer, due to the decoupling of render target size from stage size, and rendering will only render exactly the 1280 x 800 required.

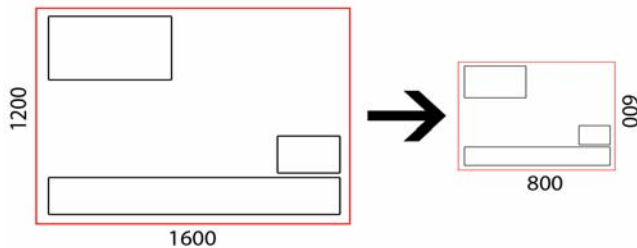


Figure 2. On the left, the conceptual stage size, at 1600 x 1200 pixels. On the right, the output stage size, maintaining relative scale, at 800 x 600 pixels. Output resolution changes, but the scene size does not.

While maintaining the appearance and relative size and position of objects is desired for some games, other games chose to scale the elements of their GUI with the change in resolution. In World of Warcraft, for example, the heads up display elements on the screen appear to become smaller as the resolution increases. A developer can easily implement this system by first designing their system at a minimum resolution. This minimum resolution may be, for example, 640 x 480. At 640 x 480, a bar that is 600 pixels wide may stretch across the bottom of the entire screen. When the resolution increases, they may want to have the bar sit in the middle of the screen, but have all of the elements within the HUD appear to shrink with the resolution, increasing the amount of screen space for the game. If, when the resolution is changed, the developer changes both the output render size and the stage size in sequence, this affect is easily achieved. A 48 pixel tall bar that was once 1/10th of the screen at 640 x 480 resolution will become 1/20th of the screen at 1280 x 960. These development choices are easily solved with a single model.

This same solution can enable developers to work on any coordinate space they chose. If a developer is more familiar with a 0 to 1 coordinate space, they can easily set the stage size to 1 by 1, giving them a potentially familiar coordinate space. It also supports the various scaling qualities, screen size changes, and most importantly allows developers and designers their freedom of choice in implementation and promises consistency through device sizes.

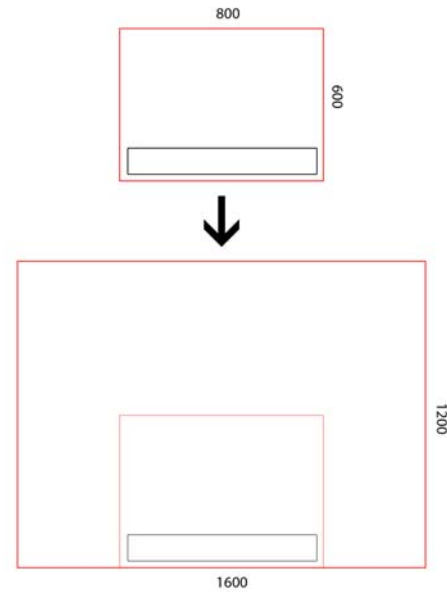


Figure 2. Illustrating how changing the stage size can increase resolution but not scale the elements, such as in World of Warcraft. The top screen illustrates the low resolution window, while the bottom screen illustrates the scaled stage size and resolution. Output resolution and scene size change together

5.2.2 Changing the Aspect Ratio

Changes in aspect ratio can affect the layout of elements on the screen. Elements that sit at the edge of the screen at a wide, high definition resolution like 1920 x 1080 become clipped from the edge of the screen at a narrow aspect like 1600 x 1200.

To solve this problem, all elements have an alignment. The alignment of an element describes where the element should sit relative to its parent. If an alignment of an element is set to horizontal and vertical center, when it is positioned at 0,0 it will remain in the center of the screen. No matter the change in aspect ratio, the element will always remain in the center of the screen. Should an element be aligned horizontally to the left, however, if the x component of the elements position is set to 0, the element will stay anchored to the side of the screen. When the aspect ratio changes from a wide screen resolution to a more narrow resolution, this element will always remain on the side of the screen, and the offset of the x component will remain consistent throughout aspect ratio changes.

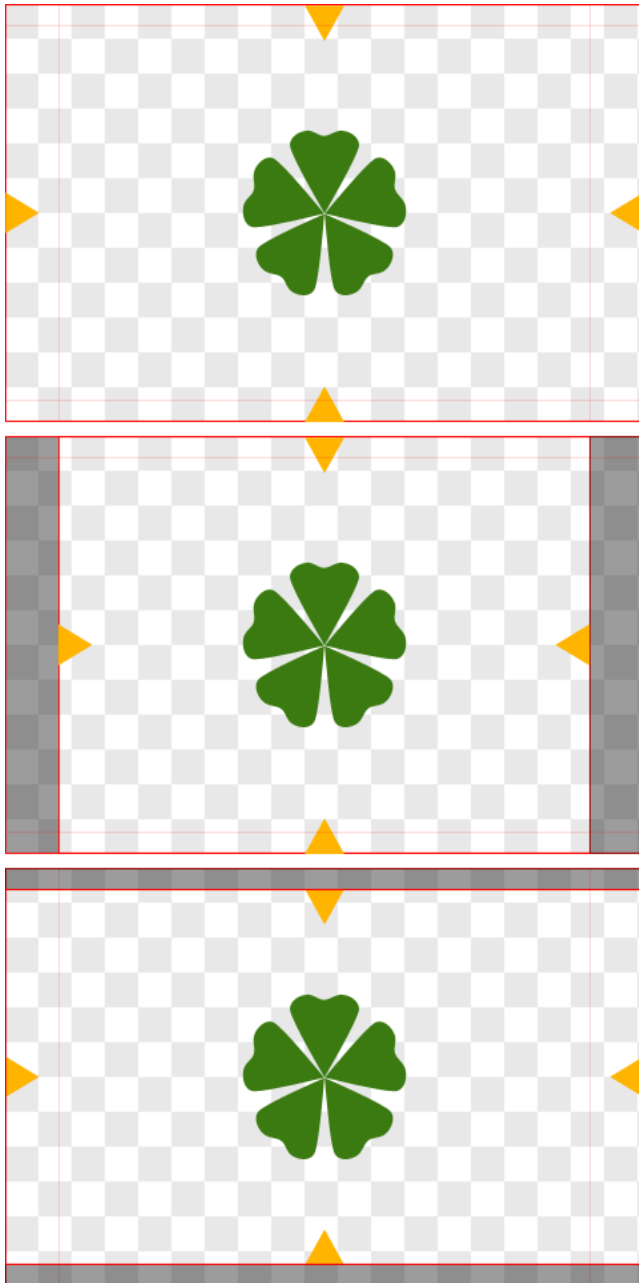


Figure 3. Illustration of changing aspect ratio but maintaining object scale and having desired elements stay on screen. The top image illustrates a 16:10 aspect ratio of 1920 x 1200. The middle image shows a 4:3 aspect ratio of 1600 x 1200. The bottom image shows the same scene again with an HD aspect ratio of 16:9 at 1920 x 1080. Background elements that are centered and sized at 1920 x 1200 clips appropriately and yellow elements that are aligned to the sides stay on the screen. All elements maintain appropriate scale. The only change made between them is on the scene size.

For all elements within the GUI, alignment affects an object to the bounds of its parent. If an object is parented to the stage itself, it will always adjust itself to the aspect ratio of the scene.

5.3 Layout Template

A layout template allows for the quick description of elements on a stage. The layout template is an XML driven format that can describe several elements on a stage and all of their properties. Any objects described through the layout template are accessible through the Scene by their string index at a later point in the application. The layout template is simply a tool for developers to use to accelerate their development process.

In order to use a layout template, it must be first loaded into the GUI core. This process will just load the XML data into the system but will not create any elements. This gives developers control over when file loading occurs.

```
Gui->LoadLayoutTemplate(path)
```

After a template has been loaded into the GUI core, any scene can enable the template by using the path that it was created with. The path serves as a primary key to the template.

```
MyScene->ActivateLayoutTemplate(path)
```

A scene can similarly deactivate a layout template, which will remove all elements from the scene that were created with the LayoutTemplate

```
MyScene->DeactivateLayoutTemplate(path)
```

Layout templates not only define elements in a scene through XML, but they are also able to reference resource sheets that may need to be loaded in order to support the layout. There is an intermediate step that must occur in order to load the resources described in a template.

```
MyScene->LoadLayoutTemplateResources(path)
```

This intermediate step is available to once again give developers control over when resources get loaded into the GUI system.

5.4 Animation of Elements

In order to animate elements within a scene, a tweening system is implemented. The tweening system is simply a way of transitioning between one value to another over time. What value, how long, what style and on what property the tween executes is up to the user.

Tweens are managed by the scene that controls the element that is being tweened. If a scene is paused, any tweens on the scene will similarly pause. Tween instances are also deleted by the scene when they are done being used, so there is no responsibility for the developer to manually manage their resources.

Tweens have several properties that define how they behave:

- **GUIElement*:** A tween acts on a single GUI element.
- **Property:** An enumeration that refers to any floating-point value on an element that is appropriate for tweening.
- **TweenStyle:** A tween style object is responsible for describing how the value changes over time. A tween style can be a predefined tween style (such as Linear, or EaseIn), a configurable tween style (such as a Bezier curve), or any custom class that the user creates that extends the ITween object that can manipulate a value between 0 and 1 over time.
- **Start Value:** The start value of the property
- **End Value:** The end value of the property
- **Time:** How much time the tween takes, or each loop of the tween, in seconds.

- **Delay:** A delay before the tween executes, in seconds. This delay will only apply the first time the tween attempts to execute, and will not come into play when the tween loops.
- **Post Delay:** A delay after the tween is completed, before it is flagged for removal, should the developer still require access to it.
- **Loop Count:** How many times the tween should loop. A tween can use -1 to loop indefinitely or 0 to only play once.

Tweens have several properties available to them for control:

- **Kill:** Sets the objects tween property immediately to its final value and flags the tween for deletion.
- **Reset:** Starts the tween over. This will unpause a paused tween. The time property will start back at 0. Any count of loops that have occurred will remain, so if a tween is told to loop 5 times and it has looped 4 times before it was reset, it will still only perform its action once more before being flagged for removal.
- **Stop:** This will flag the tween for removal and will also stop the object at its current state in the tween.
- **Pause:** Will stop the tween from animating but will not flag the tween for removal. This makes the tween available for resuming.
- **Resume:** This will resume a tween that has been paused.
- **ResetTween:** Similar to reset, but the reset tween function will reset the loop count of the tween to 0.

5.4.1 Tween Styles

A tween, in essence, manipulates a value between 0 and 1 over 1 second. This process can be linear, but it can also be extended to offer several unique graphical experiences.

The GUI system comes with several predefined tween styles that can be used, as well as two basic mathematical functions that can be customized and used by any developer.

There are a total of 41 predefined tween styles, borrowed from the open source Tweeners project [8]. These styles range from cubicEaseIn functions to elastic and bouncing functions. The elastic and bouncing functions demonstrate that it is acceptable to go beyond the 0 to 1 bounds of the graph.

The GUI system also comes with two classes that can be used to define a curve over time: the TweenHermite class, which constructs a Hermite curve [9] based on end-point tangents that can be dynamically described to the tween when it is created, and the TweenBezier, which defines a Bezier curve [10] from control points along a curve for manipulation. The TweenHermite and TweenBezier classes are both extensions of the ITween class, and lay the foundation to future developments of unique tween styles that can be easily implemented into the GUI system.

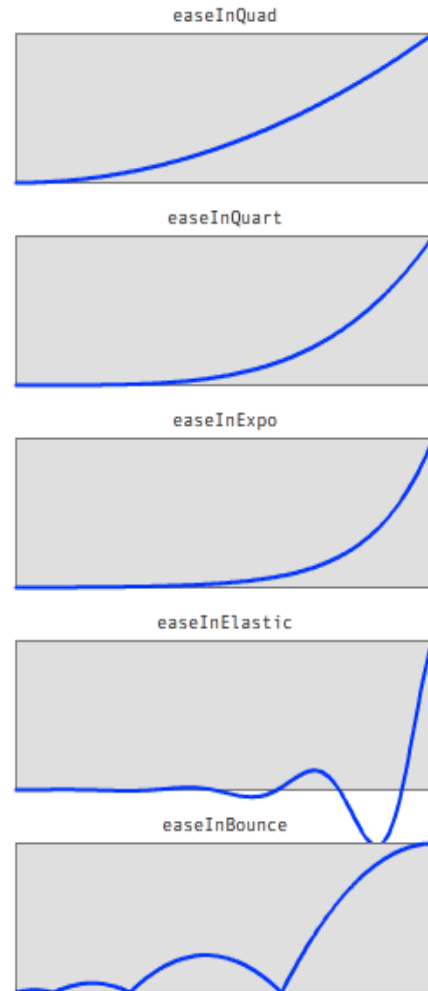


Figure 4. A selection of tween styles in the GUI system. The tween styles allow for values to go below 0 and above 1. Image from the open source Tweeners library [8]

6. RESOURCE MANAGEMENT

The GUI system implements its own resource management system. While, as previously described, GUI elements are able to use DirectX texture resources handed to them from the application, it also manages its own resources that are specific to the GUI system and would not be used in other systems in the game.

The GUI system maintains a single TextureManager throughout all of its scenes. With this, any one scene can use the same texture file that another scene may also be using.

The GUI system maintains resources internally in order to perform GUI specific optimizations and to make implementation easier for the developer.

All GUI texture resources are handled with an object called a TextureSheet.

6.1 Texture Sheets

A TextureSheet is a texture resource coupled with information describing the UV coordinates to describe cells within the sheet.

A texture sheet, for example, could be a sheet of face pictures. A GUI panel that wants to be texture with a smile face from the TextureSheet would receive the following call:

```
MyPanel->SetTexture("Faces.jpg", "Smile");
```

The call to the panel element tells the panel to use the TextureSheet "Faces.jpg" and to use the cell "Smile" from that sheet. If the TextureSheet has been appropriately loaded into the system, the panel will display just the smile face from the Faces.jpg file.

6.1.1 Loading a TextureSheet

In order to load a TextureSheet into the system, the developer would need to call the TextureManager and call the CreateTextureSheet function:

```
CreateTextureSheet("Faces.jpg", 512, 512);
```

The call to create a texture sheet requires the string name to the image file that is to be loaded. This call will return a pointer to the texture sheet object that is created. The resource "Faces.jpg", however, has not yet been loaded. At this point, only the TextureSheet class itself has been generated. The texture sheet that has been generated has been given the unique string identifier "Faces.jpg" and can be retrieved at any time through the texture manager. The resource on the texture sheet can be loaded at any time by using the LoadTextureSheetResource function on the texture sheet. Calling this function will load the file "Faces.jpg" into the system. It is up to the developer to decide when this loading will occur.

Once the texture sheet is created, whether or not its resources have been loaded yet, cells can be added to the sheet, describing what regions of the image correspond to the desired sections of image. For the example of faces, which may be a 512 x 512 pixel image with four 256 x 256 faces on it, one may want to create four different cells on it for each face. That can be done by calling the CreateCell function on the TextureSheet:

```
CreateCell(0, 0, 256, 256, "Smile.jpg");
```

The call to create a cell takes an optional string identifier to the cell as well as the upper-left hand corner of the cell and the cell's width and height, in pixels.

6.1.2 Using a Texture Sheet

Assigning a texture to a GUI panel can be done in several ways. The function signatures are listed as follows:

- SetTexture(SheetName, CellName)
- SetTexture(SheetName, CellIndex)
- SetTexture(Sheet*, CellName)
- SetTexture(Sheet*, CellIndex)
- SetTexture(SheetName)
- SetTexture(Sheet*)

A TextureSheet can be assigned by its string name for ease of use, which will cause the panel to retrieve the TextureSheet from the TextureManager, or by passing the object to it directly for

performance. A cell name or index can be used as well. Cell names are optional in defining the cells in a TextureSheet. Complex elements, like an animated texture with several cells, may not want to define a string name for each cell. In this case, a simple integer index can be used.

A texture can also be set to a panel with a cell provided at all. By default, this will cause the TextureSheet to default to its first cell index. A TextureSheet can be created without specifying any cells at all, which will by default create a 0-indexed cell that takes up the entire width and height of the texture. By creating the default texture and allowing panels to be set with just a sheet and no cell, developers can quickly treat a texture sheet as a single texture without any concept of cells, facilitating the development process.

6.1.3 Panels Response to Texture Assignment

When a panel is assigned a TextureSheet and corresponding cell, the panel size immediately becomes the width and height, in pixels, of the cell that was assigned to it. The panel width and height in pixels is not a property that can be controlled by a developer using the system. A panel will automatically assign itself the width and height of the panel to preserve size. A developer will never need to resize a panel based on the size of the texture being assigned to it. Should a developer chose to change the size of a panel on the screen, they are able to do so through the scale property of the panel. The ScaleX and ScaleY properties operate on the size of the panel. If a panel is already 256 pixels tall and has a Y scale of 2, it will effectively be 512 pixels on the screen. If the panel changes its texture to a 200 pixel tall cell, then it will show up on the screen at 400 pixels. The scale of an element stays with the element even after the texture has changed.

6.1.4 TextureSheetTemplates

TextureSheets have a similar template structure to the previously describe Layout objects. A TextureSheet can be defined through an XML file, describing the resource the TextureSheet uses and the cells of the TextureSheet. The TextureSheet template can be loaded similarly to LayoutTemplates, using:

```
LoadTexSheetTemplate("Faces.xml");  
LoadTexSheetTemplateResources("Faces.xml");
```

6.2 Texture Sets

Previously, the GUI system defined a means of handling multiple resolution outputs using the same coordinate space. While rendering to a lower resolution target appropriately calculates only the pixels needed for the output, the textures stored in video memory remained at their native, high resolution. Games will often want to support multiple texture resolution sizes for graphics in their games in order to appropriately scale to the hardware of their users.

Texture Sets help solve the problem of swapping in and out multiple resolution textures in the Resource Manager while freeing the developer from manually switching their textures on all panels in the GUI.

A TextureSheet object has already been defined as having a single texture resource file and one or more cells that represent regions on that texture. A single TextureSheet can, in fact, have multiple resources associated with it. When a TextureSheet is created, the resource that it was created with is automatically assigned to TextureSet 0. A texture sheet can add a second resource and be given a TextureSet number to associate that resource with.

In the example of the Faces.jpg TextureSheet, a high resolution faces image might be a 512 x 512 image of four faces. A second texture set can be added to this TextureSheet of a different resolution image of the same picture. This can be done with the following call:

```
AddTextureSet("Faces_256.jpg", 1);
```

Once the lower resolution image is added to the TextureSheet, it will be stored but the image itself will not be loaded.

The GUI core is responsible for receiving calls to globally change texture sets on objects. If the GUI is told to use TextureSet 1 across all resources, the TextureSheet will, behind the scenes, unload its original resource and load the texture resource described at set 1. If the TextureSheet does not have a TextureSet at index 1, it will default to the next lowest set available.

The TextureSheet preserves its cell information and all cell regions appropriately translate to the new texture set. Now, however, a lower resolution image is being stored in video memory. Should a panel be set to use a texture from the new texture set, even though the cell region may only be 128 pixels in size rather than 256 pixels square, the GUI panel will size itself to the original definition of the cell.

This becomes valuable in maintaining a single coordinate system and series of transformations across the entire application. If a game is designed to operate at a high resolution, and all elements are positioned according to their high resolution versions, if the resolution changes the GUI system is able to both update the output size of the scene and reduce the textures being used to a size that is more appropriate to the new resolution while remaining ultimately transparent to the developer implementing the system.

There is no fixed implementation of TextureSets. This allows the developer to implement the system however it may work for them. TextureSets can be used to work with resolution changes, but can also be used in others ways (for example, changing themes on objects by describing an element but having four different color themes on four different texture sets)

7. INPUT

The GUI system is not responsible for communicating with input devices. It can, however, be fed input by the application. The GUI system is responsible for handling two types of input:

- Keyboard Input
- Mouse Input

7.1 Mouse Input

The GUI system can be fed mouse input from the application. The GUI system will need to know the position of the mouse, in normalized form from the window, with (0,0) being the top left of the window and (1,1) being the bottom right, and a boolean indicating whether or not the mouse button is pressed. The GUI system then feeds this input in through any Scenes that are mouse enabled. A scene will need to be aware of its position on the screen. By default, the Scene will assume that it is taking the full screen. If the scene does not take up the full screen, it will need to be told what region of the screen it is on, from 0 to 1, in order to appropriately scale the mouse input to its elements. This can be

done using the SetScreenSpaceCoordinates(x1, y1, x2, y2) function on a scene object.

When a scene receives mouse input, it performs a barycentric triangle test [11] on mouse enabled panels. If an element tests true for the mouse input, it can inform the application that the mouse is over it.

7.1.1 Communicating with the Application

Communication with the application occurs through an observer pattern [12]. The observer pattern lets the GUI system operate independently of the application. For a class to listen for mouse events on an object, it will need to extend the IGUIMouseListener class and implement the IGUIMouseEvent function and listen to elements for mouse input through the AddMouseListener function. The IGUIMouseEvent function will return a IGUIMouseEvent object.

The IGUIMouseEvent Object contains the following information:

- The pointer to the element that received the input event.
- The event type. This is a constant that defines whether the mouse entered the object or left the object through an enumeration.

Using this information, a GUI system is able to report to the application when mouse events have occurred relative to the GUI system. It is the applications responsibility to appropriately respond to these events by changing the graphics of the elements as they are clicked or receive mouse over events and mouse out events.

7.1.2 3D GUI Scenes

GUI Scenes that serve as textures on 3D objects are a unique case. The mouse events that occur on a 3D surface are distorted by the perspective effect produced by the camera. If a developer was to work mouse input onto a 3D GUI scene, they would want to first disable mouse inputs on the scene. As a next step, they would need to, at the application level, determine where on the UV map the mouse is hovering. This can be done with several techniques, including ray casting, however this is not the responsibility of the GUI system. The GUI system does, however, implement the functions that could create such functionality. Scenes have a function to directly inject mouse input given a set of coordinates. This input only applies to the scene where the input is given and will bypass the mouse input from the GUI. This enables mouse interaction on 3D surfaces.

7.1.3 TextFields and MouseInput

TextFields are special cases within the GUI system where the system will internally respond to mouse input. MouseInput on a TextField that has been flagged as selectable will automatically handle text highlighting on the text field. Text highlighting is determined by the position of the mouse with regards to the character glyphs on the text field. Before any highlighting occurs on the text field, the text field object will need to be the center of the GUI input focus. Once the text field is registered as the GUI focus, which must be done manually by responding to input events on the text field object, the object receiving highlighting will internally record and store the button state of the mouse input being fed to it and will highlight characters according to the systems put in place by the TextField object.

7.1.4 Performance Considerations

Should performance become an issue, the developer has the option of limiting how frequently it feeds the GUI system input from the mouse.

7.2 Keyboard Input

The keyboard input system works similarly to the mouse input system in that the GUI system itself does not check for device input itself but instead must be fed input. Input should come in the form of an ASCII character representing the key that has been pressed on the keyboard.

TextFields are the only events in the GUI system that receive keyboard input in this manner. Any extensions to the system are to be implemented on a separate layer. Tab ordering, for example, could be handled on a separate level from the fundamental GUI system.

The GUI system can only have one element that has keyboard focus at a time. The element that has keyboard focus is to be set by the external application. If the developer wants a TextField to receive focus when it is clicked, the developer must implement a system that listens for mouse input on the text field and then instructs the GUI to set the current keyboard focus object to be the selected object.

Keyboard input into a text field will natively handle backspace functions, but all other characters will enter the TextField unless the TextField is instructed otherwise.

When the text in a TextField changes, the TextField object will call any objects listening for a KeyboardEvent. The KeyboardEvent object contains the following properties:

- The pointer to the element that received keyboard input.
- The last key entered on the TextField object.

Using the pointer to the TextField object, a developer can retrieve the text that is currently in the TextField and use it accordingly.

8. RENDERING

8.1 Rendering Scenes

As was previously described, all scenes are rendered to independent render targets. These render targets can be either texture resources or direct rendering to the device.

8.1.1 Device Rendering

If a scene is configured to render directly to the device, it will use the device object that the GUI system was initialized with. If the Scene object is set up to render to a device, the Scene object will automatically render every frame, whether or not the Scene is flagged as "dirty", under the assumption that the back buffer is being cleared every frame.

8.1.2 Texture Rendering

When the GUI system is rendering to a texture, it only needs to render when the scene is dirty. The texture, which is generated by the GUI system, will only need to have its back buffer cleared when the scene is flagged as dirty.

If a Panel within the scene is using a texture outside of the GUI's resource management system, the Scene will not know when the texture has been refreshed and will require manual intervention. If an external texture has been provided for a GUI panel, the GUI panel can either have an `FlagAsDirty` function called whenever the external texture is updated, or can have an `AutoUpdate`

Boolean set to true, which will flag the Panel as dirty every frame and require the Scene to re-render itself on every update.

8.2 Rendering GUI Elements

All elements within the GUI system are simple quads. Text glyphs are a series of polygonal quads and all panels quads as well. Because of this, the GUI system uses a single vertex buffer to render all scenes with.

When the GUI system gets told to render, the GUI system first passes this full screen quad to the video card. After the vertex data is on the card, each scene goes through its elements and determines if it needs to be rendered. Should the scene need to be rendered, each element renders itself by simply passing its transformation matrix, color information and texture properties into the card and rendering with the same vertex buffer. This helps performance in limiting the amount of redundant data that needs to be sent across the board at each render.

9. IMPLEMENTATION AND FUTURE WORK

All of the systems describe above will be implemented in the Masters Capstone of 2009 at the Rochester institute of Technology.

The GUI System proposed is a foundation of what could possibly become a fuller, more complete user interface system. It aimed to solidify the basic activity that a GUI system would need to give full creative freedom to developers and still enable rapid development and solve several problems that users may face in designing user interface systems.

The system as described has opportunity to be built upon, either in the code itself or in a system that implements the GUI system. Classes can quickly pull together the functionality provided to create buttons and other advanced widgets, as well as integrate more sophisticated UIML markup systems.

Improvements that could be made to the current system as it include resource batching of texture sheets into single sheets to improve performance, a scripting interface, and most importantly a masking system. Without a masking system, several interface elements cannot be created. A masking system would enable incredibly advanced widgets to be built around the proposed architecture.

10. REFERENCES

- [1] T. Kruithof, Rafael Bidarra. "Glaze: A flexible GUI engine for video games." 2006. Internet: <http://graphics.tudelft.nl/~rafa/myPapers/bidarra.siacg2006.pdf>
- [2] A. H. Jorgensen. "Marrying HCI/Usability and Computer Games: a Preliminary Look. ACM. 2004.
- [3] Unknown Author. "GUI Library for Managed DirectX Applications." 2006. Internet: <http://www.codeproject.com/KB/directx/OdysseyUI.aspx>
- [4] Adobe Staff. "Adobe Livedocs: MovieClip Class." Internet: <http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/display/MovieClip.html>
- [5] Apple Developers. "ATSUI Programming Guide: Text Measurements." Internet: <http://developer.apple.com/documentation/Carbon/Conceptu>

- [al/ATSUI_Concepts/atsui_chap2/chapter_2_section_2.html#/apple_ref/doc/uid/TP30000028-TPXREF135](#)
- [6] Adobe Staff. "Adobe Livedocs". Internet:
<http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/>
- [7] Carnegie Melon University. Panda 3D Game Engine.
Internet: <http://panda3d.org/wiki/index.php/DirectGUI>
- [8] A. Debert, N. Chatellier, Z. Fernadno. "Tweeners". Internet:
<http://www.google.com/search?q=tweener&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-a>
- [9] N. Pipenbrinck. "Hermite Curve Interpolation." Mar. 30, 2006, Internet: <http://www.cubic.org/docs/hermite.htm>
- [10] N. Pipenbrinck. "DeCasteljau Algorithm." Sept. 19, 1999.
Internet: <http://www.cubic.org/docs/bezier.htm>
- [11] "Point in Triangle Test." Internet:
<http://www.blackpawn.com/texts/pointinpoly/default.html>
- [12] J. Hermann. "Callbacks in C++: The OO Way." Feb. 16, 2000. Internet:
<http://www.codersger.de/mags/cscene/topics/misc/cs7-04.xml.html>
- [13] D. Kieras. "User Interface Design for Games." University of Michigan. Internet:
<http://www.eecs.umich.edu/~soar/Classes/494/talks/User-interfaces.pdf>
- [14] P. Chu. "It's the User Interface, Stupid!" Internet:
<http://www.technicat.com/writing/ui.html>
- [15] "UIML: User Interface Markup Language." Internet:
<http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf>

Behavior Trees in Game AI

Andrew Galante
Rochester Institute of Technology
Rochester, NY
585-749-3970, USA

drew7@mail.rit.edu
drew7@mail.rit.edu

ABSTRACT

Herein is described the methods to be explored in game agent control. Behavior trees are examined as an algorithmic solution to behavior definition for artificial agents.

Categories and Subject Descriptors

D.3.3 [Algorithms]: Artificial Intelligence applications to games – *behavior tree, control model, agent behavior.*

General Terms

Algorithms, Management, Performance, Design.

Keywords

Behavior Trees, Agent Control, Agent Specification

1. INTRODUCTION

More and more, artificial agents have been called upon in games to provide a more immersive, and ultimately, more enjoyable experience. Games such as “Halo 3”[5] and “Left 4 Dead”[9] rely on artificial agents working for and against the player to provide gameplay enrichment and more immediate engagement for the player.

2. PROBLEM DEFINITION

A Behavior Tree is a hierarchical structure detailing the desired behavior of an artificial agent. It consists of a high level task at the root, which is composed of smaller subtasks which in turn are composed of smaller tasks. The leaves of this tree are composed of atomic behavior units which may be executed directly by the agent. Decision making and error handling are handled automatically by the tree structure, which allow the agent to fall back to alternative behaviors in exceptional circumstances.

Behavior Trees are a fairly new technique in designing game AI, so this research will focus on their implementation, usage and potential pitfalls in a tank combat game.

3. SIGNIFICANCE OF THE PROBLEM IN INDUSTRY

Generating interesting and meaningful AI is a difficult problem mainly because such behaviors are relatively complex. It takes time to draw up and implement the behaviors desired from an AI. Tools and frameworks that expedite this process while reducing the chance for errors would be very useful.

The behavior and ultimately, game performance of artificial agents in usually pales in comparison to humans. To counter these deficiencies, developers have employed tactics such as increasing the agents' abilities and/or numbers[1,2]. In some cases this is considered cheating, for example, if agents are presented as human analogues and still given abilities human

players wouldn't have. However, human players can still easily learn their behaviors, and if these bonuses aren't given to the AI, they would be quickly defeated.

4. PRIOR WORK AND APPLICABLE APPROACHES

4.1 Finite State Machines

Traditionally, artificial agents in games are implemented using the Finite State Machine technique[1]. This mirrors the Finite State Machine of computer science, and produces agents with simple, reliable behavior. An example FSM can be seen in Fig. 1. It depicts a state machine for opening and closing a door. This might be used as part of a larger state machine for agents that need to deal with doors. States represent beliefs the agent has about the world, while transition conditions represent possible actions the agent can take from a given state. Entry actions are actions the agent must execute upon entering a state.

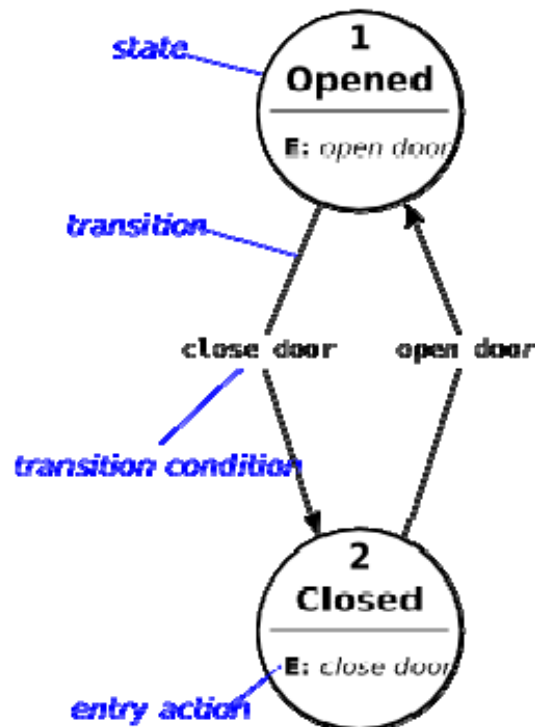


Figure 1. A Simple Finite State Machine

The problem with FSMs is that large and unwieldy state machines would be necessary to model complex behavior. This is mitigated some by using Hierarchical State Machines[1]. These extend the base FSM technique by implementing each state in a larger machine as a state machine itself. Even so, the number of state

transitions and the interplay between hierarchical levels can be difficult to manage.

4.2 Decision Trees

Decision Trees are an AI technique that enumerates possibilities[2]. Conceptually, they are similar to flowchart diagrams, but impose a strict hierarchy to them. The root decision for the tree is a conditional of some element of the game state. Its left and right children are subtrees applicable when the condition is true and false respectively. For example, the root node could contain “Is the player nearby?” If the player is, processing proceeds to the left child which may be “Shoot.” If the player isn’t, processing moves to the right child which may be a more complicated subtree.

Decision tree learning[2] is a method used to construct decision trees from observational data, and would be interesting if applied to game AI. However, it won’t be explored here.

4.3 Planning

Planning further extends decision making by being able to make decisions before acting upon them[1]. Planning algorithms have been researched for some time. STRIPS, the Stanford Research Institute Problem Solver, was designed to be able to achieve goals by searching for courses of action in a set of “world models.” Courses of action consist of lists of actions that cause the world state to transition from model to model, until a world state is reached that satisfies the goal.

Hierarchical Task Networks[5] are a simplification of this that require world models and tasks to be laid out in a hierarchy rather than a directed graph. This hierarchical nature makes them functionally similar to decision trees, to the point where the same techniques can be used on them. In this case, planning can be simplified to a search of the tree; a binary search in simple yes/no decision trees, or a more complex heuristic search in n-ary trees[2]. By designing these trees in a modular way, agent behaviors can be directly encoded in their structure.

5. DEFINITION OF ATTEMPTED DISCOVERY

Behavior Trees combine elements from planners, decision trees and finite state machines[5]. A behavior tree is composed tasks, with the highest-level task at its root. When a task finishes execution it returns success or failure to its parent to allow replanning to occur. A behavior tree can be thought of as a finite state machine where states correspond to tree nodes, and transitions correspond to branches connecting the nodes. States become systematically organized, and transitions are limited to subtree regions, allowing them to be treated as a single logical domain. This greatly reduces the amount of complexity the tree designer needs to manage, even if it increases the total number of necessary states.

5.1 Operation

To be a useful behavior tree, the root node should be an aggregate node, that is, one that contains other nodes as children. Among the aggregate node types are sequence nodes and selector nodes. For a sequence node to execute correctly, all of its children must execute successfully. For a selector node to execute successfully, only one of its children must do so. These correspond to the

logical “AND” and “OR” operations. At the leaves of the tree are atomic tasks such as sanity checks or actions the agent can perform. Below (Figure 2) is a sample behavior tree for a tank agent. This agent has the behaviors “Move to the base and defend yourself,” represented by the left subtree, and “Escort the Player,” on the right. Potential behavior tree nodes are listed in Table 1.

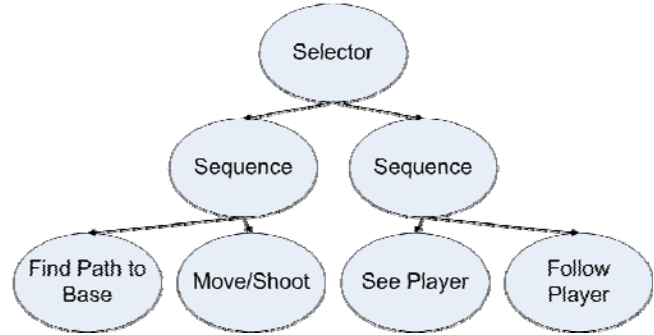


Figure 2. Sample Behavior Tree

Table 1. Proposed Node Types

Movement/Firing	
Move/Shoot	Move along the current path, shoot at targets
Flock/Shoot	Move with the group, shoot at targets
Follow Player	Follow the Player and shoot at targets
Target Heading	Aim toward given point
Target Unit	Aim and track given enemy
Join Group	Join a larger group/flock of units
Leave Group	Leave the current group
Path Finding	
Path to Cover	Find a path to the nearest suitable cover
Path to Location	Find a path to the given location
Path to Heading	Find a path to the given relative location
Path to Base	Find a path to the unit’s base
Checks	
Can See Player	True if human player is visible
Can See Enemy	True if an enemy agent is visible
Can See Friendly	True if an ally agent is visible
Decision Making/Planning	
Sequence	Run child nodes left-right until one fails
Selector	Run child nodes left-right until one succeeds
Learning Selector	Like Selector, moves successful node to front of list
Planning Selector	Estimate costs, search for plans on nodes
Other	
Script	Run custom script asynchronously

Some of these nodes imply the use of other algorithms generally associated with the field of AI. The path finding nodes all imply

the use of A* to generate path. Therefore, a number of A* optimizations will be employed, such as simplifying the path search space[8]. The movement nodes can make use of a technique for making path following appear more realistic[7].

5.2 Planning

The hierarchical structure also makes it easy to search for planning. Since selector nodes successfully execute only one of their children, they are the only nodes that are affected when making decisions. Encoding decision information inside the larger tree this way makes it easier to design the tree – when executing plans only these nodes need be involved. When searching the tree for plans, all nodes are involved as they have to supply their cost of execution to the planner.

5.3 Input

Finally, the behavior of an agent must be flexible enough to react appropriately to unforeseen events – to allow a portion of the behavior tree to subsume over another. The proposed solution to this problem is to introduce the concepts of “input events” and “input stack.” These concepts are similar in spirit to their counterparts in windowing systems, in that an entity registers what events it should receive in the input stack, and is notified when those events fire. Examples of these input events can be found in Table 2.

Table 2. Behavior Tree Input Events

Enemy in Range
Target in Range
Under Fire
Damage incurred
Ally Under Fire
Enemy Player Arrived
Friendly Player Arrived
Objective Under Attack

When a node is processed, the input events it should listen for are pushed onto the input stack, along with the desired termination status. Each update, the input stack is processed from top to bottom (most recent to least recent). If any of the listed events fires, the current node is forced to terminate with the specified termination status. This signal propagates back up the tree until an aggregate node is reached that can process the signal.

6. APPLICATION TO THE CAPSTONE

For the Capstone project, the behavior tree technology described above will be the target implementation for NPC intelligence. In the worst case, the input stack and planning portions of the algorithm may prove difficult to implement. The base behavior tree implementation already exists in the AI technology demo, thus it can be merely expanded with more behaviors if necessary.

7. CONCLUSION

Behavior Trees are a relatively new technology when applied to game AI that I think show promise and are worthy of further study. Research will be performed in applying this technique full-scale for NPC units in a multiplayer tank battle game.

8. REFERENCES

- [1] Schwab, B. 2004. AI Game Engine Programming. Charles River Media, Boston.
- [2] Millington, I. 2006. Artificial Intelligence for Games. Morgan Kaufmann Publishers, New York.
- [3] Orkin, J. 2006. Three States and a Plan: The AI of FEAR. Game Developers Conference 2006.
- [4] Fikes, R. and Nilsson, N. 1970. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence Group, Stanford Research Institute.
- [5] Champandard, A. 2008. Behavior Trees for Nest-Gen AI. <http://aigamedev.com/premium/presentations/behavior-trees>
- [6] van der Sterren, W. 2001 Squad Tactics: Team AI and Emergent Behaviors. AI Game Programming Wisdom, pp. 223-246.
- [7] Pinter, M. 2001 Realistic Turning between Waypoints. AI Game Programming Wisdom, pp 186-192.
- [8] Tozour, P. 2001. Building a Near-Optimal Navigation Mesh. AI Game Programming Wisdom, pp 171-185.
- [9] Mitchell, J. 2008. Connecting Visuals to Gameplay at Valve. Montreal International Game Summit, 18 Nov. 2008. http://www.valvesoftware.com/publications/2008/MIGS08_ConnectingVisualsToGameplay.pdf
- [10] Russel, S. and Norvig, P. 2003. Artificial Intelligence, A Modern Approach. Prentice Hall, New Jersey.
- [11] Sun, T. 6C BCE. The Art of War. Translated to English 1910, Giles, L., M. A. [http://en.wikisource.org/wiki/The_Art_of_War_\(Sun\)](http://en.wikisource.org/wiki/The_Art_of_War_(Sun))

Neural Networks

Joshua S. Gilpatrick
Rochester Inst. Of Technology
Rochester, NY
207-752-2318, USA

Jsq7808@rit.edu

ABSTRACT

This paper talks about the work I am going to do as part of my master's research. It proposes a method for developing an Artificial Intelligence (AI) that will provide emergent behavior at a high level. The paper will discuss the problem with current techniques, some methods to enhance these current techniques. The technique I purpose and a comparison between this technique and traditional techniques. Finally this paper will discuss the implementation details for the technique I have researched.

Categories and Subject Descriptors

D.3.3 [Neural Networks]: Applied Usage and Learning – *heuristics, learning, game*.

General Terms

Algorithms, Documentation, Performance, Design, Experimentation, Theory

Keywords

Neural Network, Artificial Intelligence, Games, Learning in Games,

1. INTRODUCTION

Neural networks enable a computer to do a variety of things including, problem solving, prediction, and pattern recognition. I intend on using a neural network to predict player behavior and evaluate threat to determine the high level strategy for our computer player. The architecture to be designed needs to be dynamic enough that it can be used as an assistant to the player and as an enemy against them.

Neural networks in games are often looked down upon as they are unpredictable, and can be slow if not implemented properly. Both are things that game developers don't want when trying to provide an engaging experience to the user. Only a handful of games have attempted to use neural networks and their use varies depending on the game. The suggested use for our game is to develop a network that can learn to develop strategies based on an influence map.

This use for neural networks in games has been theorized but never used in a game that has been published. While our plans our not to directly publish this game it would be nice to have an implementation that can be taken to future employers and show a contribution to the industry.

2. PROBLEM DEFINITION

Humans are good at noticing patterns. It has been said that we often play games only until we master the pattern in the game, when we arrive at this stage we become bored with the game and move onto the next one. One of the goals of this project is to

attempt to create a dynamic AI that even when the player plays the same level multiple times it will feel like it is a different battle.

Another problem is the generalization of information provided to develop an AI that engages the user in an intelligent or tactical manor. All too often we come across games where the NPC units simply charge at the user or have a set of scripts that run and provide the same results in a limited number of situations.

The final problem is the use and understanding of neural networks. While many researchers like to use neural networks for various pattern finding problems, game programmers often shun them. This may be because of the lack of understanding or the appropriate use of them. The things we know about a neural network are they take time to train and speed and memory consumption increase linearly with size¹.

3. PAST IMPLEMENTATIONS

This section will discuss about past implementations of an AI that controls the AI at a higher level. These implementations do not look at or deal with the issues of a lower level AI. An example of this AI is one that controls the animation system, or the fire behavior of an individual unit.

3.1 Rules Based System

A rules based system is a simple system that requires and expert to setup a series of rules that will be tested against when the AI needs to determine what it needs to do. The positives to this kind of system, are speed. This system has many early termination statements that only require the AI to evaluate a portion of the entire set.

The cons to this system are the time required to setup the rules. This kind of system often has a lot of edge cases that requires special rules to be setup for these things. Another con to this type of system is the ability to players to understand the set of rules and adapt their playing style to this. This type of system when used for high level AI is often easy to predict and counter once a user has played enough to understand the rules.

Some improvements on this are to build several trees that the rules system can evaluate. This helps add a level of complexity and prevents the if A then B situation. It now becomes if A then B,C, or D. In this case this requires a lot more overhead in coming up with these additional situations. This also does not stop the player from learning the tree it may only require them to make more iterations of play.

¹ Based on studies done for Neural Networks class using FANN network library. The study was conducted using a backpropagation network.

3.2 Finite State Machine

This type of system involves various states for the AI each state manages a specific situation and has outputs that lead to other states or the same state.

Each state determines a specific task for the AI to do. This type of system has the benefit of being conceptually and graphically easy to design and develop to build seemingly complex behaviors. The down side to this model is a player can easily learn the states and is able to determine the pattern to games that use this.

A way to improve upon this system is to build a fuzzy state machine (FuSM) this type of system is able to perform parts of multiple states at one time. The problem with this implementation is building states that work well with each other. A common example to think about for this type of system how do we define the behavior of a unit that is 25% in the run state and 75% in the attack state.

3.3 Decision Tree

This type of architecture is similar to that of a rules based system but is often implemented as a behavior action type methodology. More information about this type of structure and its application in video games can be found at [8].

The benefits of this are a seemingly different reaction for a variety of situations. Many implementations of decision trees stem multiple levels that contain decision trees.

The cons to this type of system are the amount of content for the AI director to come up with. To build a complex series of behaviors it limits the amount of content that players can use.

4. PROPOSED IMPLEMENTATION

The high level AI consists of several parts each part runs in sequence and feeds into the next part. The first part will be an update of the influence map and world information. This part will be done by the game each frame. The second part of the high level AI will be reading the influence map and passing these values to a pre-trained neural network this may execute over several update cycles. The third part of the system will be using the outputs from the neural network and passing them through a heuristic to see if they are feasible and if the network needs to be retrained. The final part of the system, will pass these values to the low level AI.

4.1 Heuristic

A heuristic will be used to determine the outputs for a given set of inputs to create an adaptive learning process. The neural network will be trained using a set up inputs that has been predefined. As the game runs the game will create various inputs and using this heuristic determines the estimated outcome for that set. This then continues to train the network while the game is running using a different heuristic to determine if the outcome was good or bad and change the training set accordingly.

After the training set has been changed the network will be trained again on this new set of input. This is to provide a network that has a seemingly complex behavior because it will adapt to situations that occur in the game.

4.1.1 Heuristic Evaluations

- Influence map total
- Number of units

- Unit locations
- Surrounding units
- Surrounding objects
- Player information
- Influence map data

4.2 Network Inputs

The inputs to the network will be the values of the influence map, in an area that surrounds a cluster of units. A cluster is defined as a number of units that are in a specific radius around the center point of those units. This influence map will accompany 100 nodes. The next inputs will be the number of friendly and the number of enemy units. This will be a factor in the influencing process to achieve the final decision.

4.3 Network Outputs

The network will output six values. The first four values are a combination of Attack, Defend, Separate, and Regroup. These values will be used in determining a high level decision tree for the lower level AI to use. The last two values will be an X and Y coordinate that tells the units what direction they should head. An example of how this data will be used is described below.

4.3.1 Example Output

.78032, .13342, .89092, .32342, 360, 120

This output states that the decision tree to be used for a specific group of AI units should be the Attack and Separate tree. This tree will then be parsed according to the unit's personality and the localized situation at hand.

The units may go to 360,120 on the map and ambush this location. Had the outputs said to regroup and go to this location they make a formation and attack this location instead.

5. GOALS

The goals of my research can be broken down into several levels. Each level makes an attempt to answer each problem defined, with a varying level of detail. I will attempt to use a neural network in our game that can define a series of strategies for ally and enemy NPC's to use.

The base level of the neural network will attempt to use a basic set of information supplied to it from the game state and determine the best strategy for the computer to take. This level of implementation answers each of the problems at a foundation level. The first problem is solved by using the fuzzyfication of the network to determine a strategy. The output of the network is not concrete meaning that if a small change in the input to the network yields a similar result it may not have the same result. This allows for emergent strategies that proved a more dynamic play experience. This base level also looks at solving the "intelligent" AI problem, however it's important to mention that because nothing is guaranteed with a neural network this may not be the case.

With the neural network trained appropriately at a high level the AI should know how to avoid getting into situations that are a loss for it. With a rules based system an expert often defines the situations and the solutions. However there are many edge cases

that need to be accounted for and can take time to program. With the appropriate training set this neural network should recognize these situations and react accordingly.

The final problem is solved in the application of the neural network. An understanding of neural networks will be required before this can be implemented. This understanding also requires and understanding of the problems games and the solutions that neural networks provide. This is a perfect merge of my two studies as a masters student.

The next level of the neural network will be to enable the network to learn on its own. Providing the neural network with basic training, then let it learn the rest as the game is being played. This will answer the first question more in-depth providing a dynamic AI that changes based on the players style of play.

6. POTENTIAL ROADBLOCKS

The first problem needed to be answered before implementing the neural network is the problem of inputs and outputs. Research was conducted on several papers about high level strategies in RTS games and first person shooter games.

Next research was conducted about influence maps and ways they could be analyzed. Finally a conclusion of a combination of inputs from an influence map and the number of units in a specific area would be an ideal input to the neural network.

Next came the problem of output, how to categorize the output of the neural network to something that can be passed to a smaller architecture that can localize the strategy and break it down into a useful set of tactics and movements.

After the foundation of questions have been resolve it will be interesting to look into how to adapt the neural network to learn. Having a network that has a complicated decision tree is a good start but is still able to fall into the trap of being too predictable. Adjusting the neural network to learn various patterns in the human behavior, and change play styles based off of these

behaviors would be an interesting next step. The problems that arise with this are, one neural networks often require some kind of supervised input, two adaptive learning may unlearn the preprogrammed values the network has learned.

7. REFERENCE

- [1] DOI=<http://aigamedev.com/reviews/top-ai-games>
- [2] Megan Vasta, Neural Networks
DOI=<http://www.cse.lehigh.edu/~munoz/CSE497/classes/MeganNeural.ppt>
- [3] Penny Sweetser, Using Cellular Automata and Influence Maps in Games.
DOI=http://www.cp.eng.chula.ac.th/~vishnu/gameResearch/sat_penny_sweetser.ppt
- [4] Timothy Adam Smith, Peta Wyeth 2001, Neural Networks in RTS AI
DOI=<http://www.innovexpo.itee.uq.edu.au/2001/projects/s369674/thesis.pdf>
- [5] Maurice Bergsma, Peter Meij, 2006 Influence Maps
DOI=<http://www.slideshare.net/mobius.cn/influence-map>
- [6] Chris Miles, Sushil J. Louis Towards the Co-Evolution of Influence Map Tree Based Strategy Game Players
DOI=<http://www.cse.unr.edu/~sushil/pubs/newestPapers/2006/cig06/cig2006.pdf>
- [7] Bobby D. Bryant, Kenneth O. Stanley, Risto Miikkulainen, 2005 Evolving Neural Network Agents in the NERO Video game.
DOI=<http://www.cse.unr.edu/~bdbryant/papers/stanley-2005-cig.pdf>
- [8] AITopics / DecisionTrees, 2008
DOI=<http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/DecisionTrees>

Multi-Threading Game AI

Nicholas A. Korn
Rochester Institute Of Technology
Rochester, NY
216-789-4607, USA
nak9193@rit.edu

ABSTRACT

In this paper, I discuss the topic for my master's research topic. I propose a method for implementing multiple threads for use on a two tiered artificial intelligence system. I take a look at options on how to split the systems for threading as well as current games which use or could use those options. Finally the paper discusses any pitfalls I can find in my implementation as well as a backup implementation.

Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management – Concurrency, deadlocks, multiprocessing / multiprogramming / multitasking, mutual exclusion, scheduling, synchronization, threads.

1. INTRODUCTION

Multi-core processor based consumer computers have become more prominent in the past few years. Having access to additional cores allows for the developers to create sections of code which can be performed in parallel with the interest of increasing performance. The system we are creating for our game consists of a high level and a low level AI system. I plan on using multiple threads in order to run both these systems in parallel with the game. The implementation will also need to take consideration systems which do not have the ability to run threads in parallel.

2. PROBLEM DEFINITION

The artificial intelligence system we are utilizing in our game consists of two pieces, a high level system which communicates with the low level. Based on the implementation of these two systems they have the ability to run at separate frequencies during game play. The challenge will be to create a usable threading system which works well with how the two AI pieces need to communicate between one another and the game.

In order to properly thread an application the developer needs to identify any points of contention between each thread. The implementation of the two AI systems will require at minimum an additional two threads on top of the main program thread. Points of synchronization between each of these threads will need to be taken into consideration. Defining what each system takes as an input from another system and what it outputs to the other systems.

One area of concern when threading is determining which systems could be writing to the same memory location at the same time. In order to protect a situation like this from occurring, a lock will need to wrap around those sections of a code preventing multiple threads from entering those areas at the same time. A key to improving performance through multi-threading will be to minimize amount of time spent waiting for locks to become available and time spent locking.

Read and write contentions may also be a problem area between the systems. This is where memory is being written to by one thread at the same time another is attempting to read that same memory. It can possibly lead to inconsistencies between systems depending on how precise they need to be synchronized to one another. Time will need to be spent determining sections of code where this will become an issue and minimize any performance hit which could occur during those sections.

Management of the AI thread during the game loop can possibly cause problems. As the game switches between states the threads may need to be terminated, put on hold, resumed, and generated. When the player switches from playing the game, or pauses, the AI threads should not be performing any work. When switching between the game, pause mode, and back to game the AI needs to have maintained its state during this period. Checking frequently for state changes could mean performance issues while checking infrequently could mean large changes in AI state. Similarly threads need to check when the game has quit so there is a clean exit.

Another problem area will be to address systems which do not have enough cores to run each system on a separate processor core. The system will need to be designed to take advantage of multiple cores but provide a solution which will work well with machines with only one. This could mean writing a second pipeline which utilizes the AI in serial as opposed to separating them into multiple threads. It could also mean simulating thread behavior in a serial manner forcing the AI to only run at certain intervals.

3. PRIOR WORK AND APPLICABLE APPROACHES

In order to split a program into multiple threads the developer needs to determine how and what will be broken down into separate threads. The intent is to split the high and low level AI systems off into separate threads for processing. There are various ways in which the developer can go about breaking these sections down for threading. The focus will be on three different ways and how they have been applied in other game development projects.

The first possibility for decomposing the project for threading is to use *functional decomposition* [Bayliss, 2008]. The idea behind functional decomposition is to take a problem (AI in this case) and break it into parts which when combined can reproduce the original function [Wikipedia, 2008]. As a part of an AI system functional decomposition would separate goal setting and actions into their own functional section. Systems which have multiple homogeneous cores work well with functional decomposition as long as the number of functional modules does not exceed the number of processing cores [Bayliss, 2008].

The game F.E.A.R. uses an AI system which is a good example of what is broken down functionally. There are two decoupled sections of the AI, one which is goal oriented and another which describes actions to accomplish the current goal [Orkin, 2006]. The two sections of the AI communicate through a section of working memory in order to allow smoother transitions between goals using the actions. This AI provides a good starting point for threading as it is already broken apart using functional decomposition. Each of those functional pieces has the ability to run on its own, as a thread, communicating through the shared memory with the other.

Another way of breaking down a program for use with multiple threads is to use *data decomposition*. The purpose of data decomposition is to divide the data into multiple segments for use in separate processing units [Bayliss, 2008]. When using data decomposition on an AI system it is natural to split the AI units into their own groups for processing. Once the units have been placed in groups a thread can take that set of data and begin processing. This form of decomposition works well on systems like the Playstation® 3 (PS3) where there are multiple processing cores (Synergistic Processing Units) which each have 256KB of local memory and thus cannot handle large amounts of data at a time.

Craig Reynolds' implementation of schools of fish, named PSCrowd, on the PS3 utilizes data decomposition in order to update fish in the system. The implementation used spatial hashing in order to split the number of AI units into separate bins for processing on separate threads running on one of the six SPUs [Reynolds, 2009]. Reynolds allowed for each bin to have read-only access to the neighboring bins based on a set radius. The implementation was able to achieve 60 frames per second (fps) while simulation 15,000 fish using minimal graphics and 10,000 with more complex graphics [Reynolds, 2009].

A third decomposition model is a combination of both the functional and data decomposition models. This is known as *hybrid decomposition*. A system utilizing hybrid decomposition will break systems apart using function or data decomposition as tasks for use by a thread. As part of an AI system using hybrid decomposition works well, allowing high level concepts to use functional decomposition and low level to use data decomposition [Bayliss, 2008]. Issues arise in using hybrid decomposition as the system has to be developed as a more general solution to cover both decomposition methods.

Valve's Source™ game engine utilizes a hybrid decomposition model [Leonard, 2007]. A pool of threads is created in order to handle tasks which are queued by the game system. The systems have the ability to generate tasks based on three levels, full system, coarse (functional), and fine grained (data) to be passed for processing by the threads [Leonard, 2007]. For AI this provides a nice interface allowing for the planning system to run on one thread and individual unit groups to be updated on others.

Valve spent a large amount of time when developing their threading utilities focusing on making the "system easy to use, hard to mess up" [Leonard, 2007]. The developers provided numerous high level functions in order to push tasks off to cores, create parallel for loops, and other things. By providing these interfaces the developers are able to approach threading segments of code on a higher level.

4. IMPLEMENTATION APPROACH

The threading system will be designed in order to allow hybrid decomposition of the AI system. In order to do so a thread pool will need to be created as well as a task management system which will receive tasks from each of the AI systems. A generic interface will also need to be created so tasks from the different AI parts will be able to submit their specific tasks to be processed by the threads. This system design will allow for the planning system to submit itself using functional decomposition and the low level AI to complete work using data decomposition.

4.1 Interfaces

4.1.1 Task

This is an interface to be implemented by classes in order for a thread to process. A thread will call this interfaces update function which takes in a void pointer to be cast for use by the function. A class implementing the task has the ability to submit one or more tasks to the manager. Additionally a task has a cost associated with it. A functional decomposition may cost 10 while a data decomposition element may cost 1. Through this method the task manager can distribute work based on the costs.

4.1.2 Shared Memory

This interface is to be implemented by classes which will need to be protected by locks before memory is written to or read from. This will give classes the ability to initialize the lock, clean up the lock, lock and unlock.

4.1.3 Double Buffered

This interface is an extension of shared memory. It provides the functionality for creating double buffered classes. It will be able to write to the back buffer, retrieve the front buffer and swap the buffers. Locks will be used to prevent the buffer from being swapped during reads and writes to the buffered data.

4.2 Structures

4.2.1 Thread Arguments

This contains event handles for sleeping, waking and terminating a thread. It also contains a pointer to the task manager in order to retrieve the next workload to be processed. A granularity value is set as well. This sets the maximum task cost a thread is allowed to accept when asking for work.

4.3 Thread Pool

The thread pool will create N-1 threads where N is the number of available processors. If the number of available processors is 1 then the thread pool will return a status saying to not use threading. The thread pool will need to receive a pointer to the task manager and will create a thread argument structure per thread. These will be passed in to the thread with the sleep event already signaled. In order to start using threads the wake even will need to be signaled. The thread pool will also have functionality to put threads to sleep, wake them and terminate.

4.4 Thread

A thread takes in a thread arguments structure. Once the thread arguments have been retrieved the thread enters an infinite loop which can only be terminated via a terminate event. The thread will check each of the events to see if it needs to process them per iteration. If no event is triggered then the thread will retrieve and process work from the task manager.

4.5 Task Manager

The task manager is used by threads and systems which assign tasks for use by threads. It implements the shared memory interface in order to allow systems to add tasks to the manager without interference. When a thread requests tasks it gives a granularity value, based on what is in the queue the manager will distribute the maximum amount of work to the thread based on the value.

4.6 AI

Both AI systems (high level and low level) will need to implement the task interface. This will allow them to assign tasks to the task manager for processing. By using hybrid decomposition the high level system will be able to submit itself as a single task while the low level system will be able to submit groups or individual units for processing.

The AI systems as well as the game world have access to an influence map. This will need to implement the double buffered interface so the game update can create a new influence map while the AI systems have the ability to read the previous state of the map.

The high level AI passes a data structure containing information about how to approach the current state of the game to the low level AI. On the low level side this should be treated as a double buffered data structure. This will allow the low level AI to work through an iteration of the high levels command before starting on the next. It prevents the high level AI from changing the course of action taken by the low level in the middle of calculations. If the high level were to pass data to the low level twice before the low level finished the current iteration then the back buffer would be over written.

As the high level and low level AI each complete a task loop they will need to re-add themselves to the task manager. If they were to be added by the game loop it could potentially lead to overflowing the task manager. Assuming the game loop runs faster than the two AI systems the task manager would not be able to get rid of tasks as fast as they were being added.

The implementation is designed in such a way that changes to the AI system should not have a large impact on how the threads operate. As long the systems utilize the threading tools which are given to them for use there shouldn't be any major reconsiderations based on AI changes.

During the game loop if the thread pool has, upon initialization, stated threading will not be used then the AI will need to run as a serial system.

5. GOAL

The goals for my research can be broken down into three parts. The main goal for my implementation is to increase the overall performance of the AI system. By utilizing threads to run individual AI systems in parallel I should be able to lower the performance hit of the AI per frame. To determine if performance has increased through threading I will have to create a baseline performance test running the AI systems in a serial manner.

The second is to create a threading system which implements well with the rest of the game engine which is run in serial and in various states. This is solved through the implementation of the thread pool which will provide access to starting, stopping and terminating the threads running in the game. This is also solved through the use of the double buffer so the game engine can

update information shared between threads with minimal interference.

Another goal for the research will be to create a system which does not cause a loss in performance on systems which do not have the ability to run multiple threads in parallel. This is solved by the use of the thread pool which will determine if the application can be run in a parallel threaded model. If a parallel threaded model is not possible it will need to run the code in serial.

6. IMPLEMENTATION PITFALLS

One of the major pitfalls of this implementation is the need for the high level and low level AI to resubmit themselves to the task manager. If the low level AI has a task which breaks down into smaller tasks using data decomposition the low level AI's main task will not know when the individual tasks are complete and can be re-added to the task manager. A possible solution for this issue would be for the low level AI to check the task count and determine if more tasks should be added at this time.

Another pitfall to this system is that it will not be as tightly linked to the AI as it could be. In creating a more generic system for use with the AI in order to perform hybrid decomposition I may not be able to squeeze out as much performance as I would when working more directly with the AI systems. If I find the system is not performing the way I anticipate adjustments may have to be made to work closer with the AI systems.

6.1 FALLBACK IMPLEMENTATION

There are two ways in which the implementation will be able to fallback. The first method of fallback will be to switch the threading system from a hybrid decomposition model to a functional decomposition model. In the case of this occurring some of the tools created by the hybrid implementation can be reused and tweaked as needed. These tools would be the shared memory and double buffered interfaces as well as the thread. The thread pool could be used in order to generate the two threads needed for this model as well as for sleeping, waking and terminating threads. The largest change between the models would be removing the task management system. Systems would not be assigned tasks during this period but instead would be given to a thread and set to run until asked to stop.

The implementation can also fall back to a single threaded model. In this model we would simulate a threaded model by updating the AI systems using different frequencies. An example of this would be to run the high level AI every 1/15th of a second and the low level at 1/30th of a second.

7. REFERENCES

- [1] BAYLISS, J. 2008. AI Architectures for Multiprocessor Machines, *AI Programming Wisdom 4*, 305-315.
- [2] CHAMPANARD, A.. 2008. Hierarchical Logic and Multi-threaded Game AI, *AIGameDev.com*, <http://aigamedev.com/architecture/hierarchical-logic-multi-threading>
- [3] CHAMPANARD, A.. 2008. Multi-threading a Simple Hierarchical Planner to Estimate Performance, *AIGameDev.com*, <http://aigamedev.com/premium/tutorials/multi-threaded-planner-performance>
- [4] CHAMPANARD, A. 2008. The Catch for Multi-threading Decision Making, Q&A with Alex

- Champanard (Video), *AIGameDev.com*,
<http://aigamedev.com/architecture/multi-threading-catch>
- [5] CHAMPANARD, A. 2007. Multi-threading Strategies for Game AI, *AIGameDev.com*,
<http://aigamedev.com/questions/multi-threading-strategies>
- [6] REYNOLDS, C. 2006. *Big Fast Crowds on PS3*, Sony Computer Entertainment, US R&D,
<http://research.scea.com/pscrowd/PSCrowdSandbox2006.pdf>
- [7] AKHTER, S. 2006. Multi-Core Programming: Increasing Performance through Software Multithreading. Intel Press.
- [8] LEONARD, T. 2006. Dragged Kicking and Screaming: Source Multicore, *Game Developer Conference, San Francisco*.
- [9] ORKIN, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. *Game Developer Conference, San Francisco*.
- [10] WIKIPEDIA. 2008. Functional Decomposition

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Gameplay Prototype Report

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

The game play prototype was completed in XNA C#, this document describes the purposes of this prototype and the things we have learned from building the prototype.

1 Goals

The gameplay prototype had several goals we wanted to accomplish. The first goal was to test how intuitive the controls would be. The second goal is to test the moving and shooting of obstacles and tanks.

2 Gameplay Prototype Requirements

This section is reserved for describing what needs to be illustrated in our gameplay prototype. Here is a preliminary list. As an example, we do not need user interface, mission switching, mission statistics. This is to test the core gameplay mechanic and make sure that what we are playing is engaging and fun.

2.1 Elements needed to test

The following elements need to be part of our gameplay prototype to make certain that our gameplay is enjoyable.

- **Player Tank Controls:** We need to be able to test all of the attributes that describe a player's tank. This includes the movement of the tank, the ability to attack with a tank, and the health attributes that describe a tank.
- **Player Tank Classes:** We should be able to test the statistics of all of the player tank classes that we have described in our game design document. This includes the Hornet, the Lancer, the Ant, and the Rhino.
- **Camera:** Our camera needs to be minimally established at the height that it is going to be set at by default in our game.
- **Environment:** We need an environment that, at the very least, provides the obstacles that one would expect to exist in the world. This is to test the actual gameplay itself, which focuses on the ability to use the environment as defense against ongoing attack and use the layout of the battlefield strategically. We also need an environment that is around the size that we are expecting one of our missions to be, to make sure that traveling around the map is not boring and tedious.
- **Objective:** We should minimally have an objective to accomplish. The objective does not need to actually do anything when achieved, but it should exist and be an understood target of gameplay.
- **Enemies:** We need to have enemies in the world that behave in the same way as our enemies do according to our design document. While these enemies may not have the finished, finalized

artificial intelligence of our proposed AI system, we should make certain that enough enemies exist in the world and provide a challenge to the user to test our core gameplay mechanics. We should also make certain that enemy behaviors are implemented and that we test multiple types of enemies and behaviors.

- **Friendly Units:** We should make sure that there are friendly units in the world and that they interact basically with the enemy units and the player. This is to ensure that friendly units are somewhat effective in partaking in the battle.

3 Implementation

The prototype was developed in XNA (C#) this allowed us to begin work in an environment that has been setup for rapid prototyping. The original prototype was going to include the following; Controls, Low Level Base Artificial Intelligence, Base set of Unit behaviors, Objectives, Unit Stats, Map Scale. With a focus on moving and shooting the obstacles (enemy players) .

The set of features implemented are; controls, unit stats, map scale, and low level AI (needs work). In the implementation phase we had a few questions that we used the prototype to answer. Some of these may still be in the code, and some were completely removed. One of these issues was the question of how we implement tracks.

Several attempts were made at this; these attempts included storing a history of tracks over several frames and redrawing them each time. This proved slow and was remove. Other methods tested were drawing a texture too two depth buffers however this proved to be invalid also. Finally this code was removed and a 2 pass system was settled upon.

4 Results

The results of our prototype are somewhat mixed. Due to some constraints we were not able to get fully conclusive results out of our prototype. It would be nice to have an AI that had some team based characteristics. This would help us better understand what about this mechanic will be fun and what needs to be improved upon. Another thing that we would like to have implemented is objectives; currently the prototype seems what void of meaning without them. However even with its shortcomings overall it was a successful prototype and told us many critical things that will alter design and implementation.

One of the things learned from the gameplay prototype is not enough screen real estate for the user to get a good feeling of how big the environment is around them. Some of the solutions to this problem are to build a minimap. The pros to building a minimap are, they are easy to build, common game

elements that players are familiar with, and they let the user know units that are just off the screen that surround the user. The cons to this type of thing are, they require another pass. Currently our engine will have several passes for lights, and other object another pass is slow and may not be necessary if an alternative solution is thought of.

Another solution would be to move the camera so the player tank is no longer in the center of the screen instead they are slightly alters to the top or bottom side, depending on their movement. This alteration will give the player a better view of the battle field ahead of them. The down side to this implementation is you have to make sure the camera is smooth in transitions from the top of the player to below the player. Another con to this type of system is you restrict the player from seeing what is going on behind them. This also does not help with the edge of the screens. So while this is a simple implementation, the cons out weights the needs for this solution.

Another thing learned from the gameplay prototype is, a complex AI isn't required to build engaging behavior. A simple AI gives the game a lot of potential to become an engaging experience for the user. An AI that acts as a puzzle against the user instead of a power against the user may provide interesting gameplay if the AI does not turn out to be good enough.

Objectives have proved to be something that is needed as a motivator to the user. The game loses value if no objectives are implemented. The player will lose a sense of motivation and stop playing. Even with the currently designed level it will be important where the placement of the objectives are and what their triggers are balanced to.

Fighting with AI units on your side, in a losing battle is something that proved engaging. In this prototype the AI is simple and does not have any complex behaviors such as team work, however with the AI their assisting the battle it adds to the gameplay as expected. Without the friendly AI there the map seems empty with a swarm of enemy units. With the friendly AI units they in this case they lead the way, this may prove beneficial in the final game and will need to undergo testing. In the current design the player leads the way.

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Technical Prototype Reports

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Mesh File Formats Technology Test

Testing solutions for a mesh file format

Goals

The purpose of this document is to research and decide upon a file format to be used in the development process of our game. The things looked at when choosing a file format are as follows:

- **Support all required mesh information:** A single mesh file should contain all information required to appropriately define a model. This information is a variable number of UV coordinates, normals, vertex position data in world space, and indices to build a triangle list out of the vertex information. Additional information that will be required is skeletal animation. A complex animation sequence is not required as we will not be implementing IK animations a simple skeletal or key frame animation will suffice.
- **Integrate with the development software:** The mesh file format should be exported from the modeling tool of choice of the development team. The development tool of choice is Maya as our artists are comfortable with this environment. The tool needs to export the file directly into a file format that can be loaded into our game. We do not want to run a post processing tool, as this may lead to version conflicts or other issues.
- **Load fast and efficiently:** The mesh file data should be easily built into a binary file format. Similarly, it should contain as little unnecessary data as possible.
- **Ease of implementation:** To render the model and animate the model this should take a minimal amount of time to implement. If the format has example code, or is well documented this will be something that is taken into consideration.

Requirements

A review of the game and the designer's needs has resulted in the following requirements of a mesh file format.

Required

1. Support of UV Texture coordinate information per vertex for texture mapping objects within the world.
2. Support of bones / joints and vertex weights for smooth binding of vertices to bones / joints.
3. Support of skeletal animations (transformations of bones / joints over time)
4. Export from Maya 2008

5. Support exportation of points that are not rendered but can be used for placing particles, or other nodes.

Among many other elements, there is a list of items that are not required for the game.

Not Required

1. Per-vertex colors
2. IK animations
3. Exportation of multiple models in one file.

Reviewed Model Formats

Collada (DAE)

The collada format is an XML based open interchange format for storing mesh data.

Benefits

- Supported Maya exporters
- Supports all mesh file requirements
- Highly general and flexible ascii format
- Easy to understand API
- Already have an implementation

Drawbacks

- Heavy-weight. Too much unnecessary data.
- Intended as an interchange format. Preserves too much of the editing applications features.
- Slow to load, requires custom conversion to get rendering.

Quake 2 (MD2)

The Quake2 MD2 file format is a commonly used mesh format for video games.

Benefits

- Many utilities to import this into DirectX.
- Already have a DirectX 9.0 working implementation.
- Project already working with it.
- Supports key frame animation.
- Well documented
- Supports key frame animation.

Drawbacks

- Does not support bone animations
- Requires custom export utilities that are hard to find or don't work properly

Quake 4 (MD5)

The Quake4 MD5 is a successor to the Quake2 MD2 format previously researched.

Benefits

- Supports key frame animation
- Supports joint animation
- Somewhat documented format.

Drawbacks

- Limited to no Mays support
- No development examples

Research Conclusion

Research shows that the simplicity of our requirements these file formats over fit our needs. Even with the prior implementation porting them over to this new engine may provide more than a week's work. The balance seems to be a fine line of developing a Maya plug-in to export to a format that is well defined and has community support, or developing the code to support a format that can be exported from Maya. In the first case it would require learning the format of the file and learning how to export from Maya. Then convert old code into something that is DirectX 10 compatible and compatible with our engine.

The later would involve slightly less work. The work would be developing the animation format that is compatible with the pre-defined format. We would need to learn this format and build the animation code. This code may be assisted by an API provided by the file format, but we risk this format being a special case for our engine.

It was decided that if, we could quickly develop a Maya export for a static mesh that this would fulfill our requirements and require approximately the same amount of time as doing this for an already defined format. The pros to this are we have a format that we understand and can manage and adapt to our needs and the needs of our engine. We do not have to conform and learn someone else's reverse engineered format, or API to get a mesh rendering in our system. The down side to this is we have to manage the format and it may not be as efficient as a pre defined format.

The models we have ordered have a binary file format requirement as part of their license. This is something that rules out the Collada format and is another benefit to building our own.

Prototype

Our prototype allows for exporting models from Maya and importing the models into a DirectX 10 environment. This prototype proves that we have the ability to do skeletal animation and support the base model format that our game requires.

The Maya exports two files, the first file is the model file and the second file is the animation definition file. Below describes the structs and the file protocol for reading and writing this kind of file. This prototype was a **success** and the information learned and the tools developed here will be used in our final project.

Mesh File format

CFHeader (struct)	
• Int	<i>Identity of this file this is always set to 2009</i>
• Int	<i>The current version of this file</i>
• Int	<i>Number of verticies</i>
• Int	<i>Number of indicies</i>
CFVertex* (struct)	
• CFVector3	<i>The position of the vert</i>
Int*	
<i>An int array that has number of indicies values</i>	

Animation File format

CFAnimationHeader	
• Int	<i>Number of joints</i>
CFJoint*	
<i>The size of the number of joints</i>	
• CFVector3	<i>Translation of the joints initial position</i>
• CFVector3	<i>Rotation the initial rotation of this joint</i>
• CFVector3	<i>Initial scale of this joint</i>
• CFVector3	<i>Initial orientation of this joint.</i>
• Int	<i>Parent joint id in order of array</i>
• Int	<i>Number of frames</i>
• CFKeyFrame*	<i>Array of keyframes for this joint</i>
• Float*	<i>Array of weights this joint affects</i>
CFKeyFrame*	
<i>The array of all the keyframes for joint n</i>	
• CFVector3	<i>Translation of the joint at this frame</i>
• CFVector3	<i>Rotation of the joint at this frame</i>
• CFVector3	<i>Scale of the joint at this frame</i>
• CFVector4	<i>Rotation tangent for the X rotation of this joint</i>
• CFVector4	<i>Rotation tangent for the Y rotation of this joint</i>
• CFVector4	<i>Rotation tangent for the Z rotation of this joint</i>
• CFVector4	<i>Translation tangent for the X rotation of this joint</i>

- CFVector4 *Translation tangent for the Y rotation of this joint*
- CFVector4 *Translation tangent for the Z rotation of this joint*
- Double *Frame number for this frame*

Float* *The array of all the weights joint n*

CFVector3

- Float *X value*
- Float *Y value*
- Float *Z value*

CFVector4

- Float *X value*
- Float *Y value*
- Float *Z value*
- Float *W value*

Extra Notes

The technology demo that went along with this paper can be found in the CFMayaPlugin and FileFormat folders of the demos directory.

Artificial Intelligence Capabilities Test

Prototyping the Artificial Intelligence system

Goals

The purpose of this technology test is to determine the feasibility and complexity of the artificial intelligence algorithms proposed for use in the Capstone game (title TBD). It tests elements of both the high-level command AI, and the individual unit AI. The command AI should be able to analyze a situation, and issue commands to the unit AI. Unit AI should be able to execute their commands to the best of their ability, according to their individualized behaviors.

Requirements

In order to be a successful technology test for the command AI, it should be demonstrated that the AI recognizes strategic situations and issues applicable commands to the units under its control. The AI needs to have the ability to assess the situation according to the influence map and determine an appropriate course of action. This action is based on some heuristics that are set in place by the developer.

The AI for units should be able to handle different tactical situations according to their individualized behaviors and commands issued from the higher level AI. In order to be practical, many units must be able to run simultaneously. The feasibility metric refers to the ease of implementation, both for the architecture and the specific implementations of each individual.

Finally, the two halves of the AI system must be able to interact with each other in a meaningful way. The high-level strategic decisions must be communicated to the individual units in a clear manner so that they can be acted upon.

Implementation

The tests for both of these AI algorithms are implemented on a C++/Direct X 10 testbed, which includes a simple terrain with obstacles. The implementation is broken down into two sections. The first we are going to talk about is the high level AI or the command section and the second section is the low level AI or the unit AI.

High Level AI

The high level AI consists of several parts each part runs in sequence and feeds into the next part. The first part will be an update of the influence map and world information. This part will be done by the game each frame. The second part of the high level AI will be reading the influence map and passing these values to a pre-trained neural network this may execute over several update cycles. The third part

of the system will be to use the outputs from the neural network and passing them through a heuristic to see if they are feasible and if the network needs to be retrained. The final part of the system will pass these values to the low level AI.

Network Inputs

The inputs to the network will be the values of the influence map, in an area that surrounds a cluster of units. A cluster is defined as a number of units that are in a specific radius around the center point of those units. This influence map will accompany 100 nodes. The next inputs will be the number of friendly and the number of enemy units. This will be a factor in the influencing process to achieve the final decision.

Network Outputs

The network will output six values. The first four values are a combination of Attack, Defend, Separate, and Regroup. These values will be used in determining a high level decision tree for the lower level AI to use. The last two values will be an X and Y coordinate that tells the units what direction they should head. An example of how this data will be used is described below.

Example Output:

.78032, .13342, .89092, .32342, 360, 120

This output states that the decision tree to be used for a specific group of AI units should be the Attack and Separate tree. This tree will then be parsed according to the unit's personality and the localized situation at hand.

The units may go to 360,120 on the map and ambush this location. Had the outputs said to regroup and go to this location, they make a formation and attack this location instead.

Low Level AI

The unit AI is implemented as a modular Behavior Tree, an experimental AI framework. A behavior tree is composed tasks, or behaviors, with the highest-level task at its root. This task has subtasks as its child nodes, and so on down to the leaf tasks. When a task finishes execution it returns success or failure to its parent to allow alternate sub-trees to be explored. Leaves of the tree are atomic tasks such as sanity checks to actions the agent can perform.

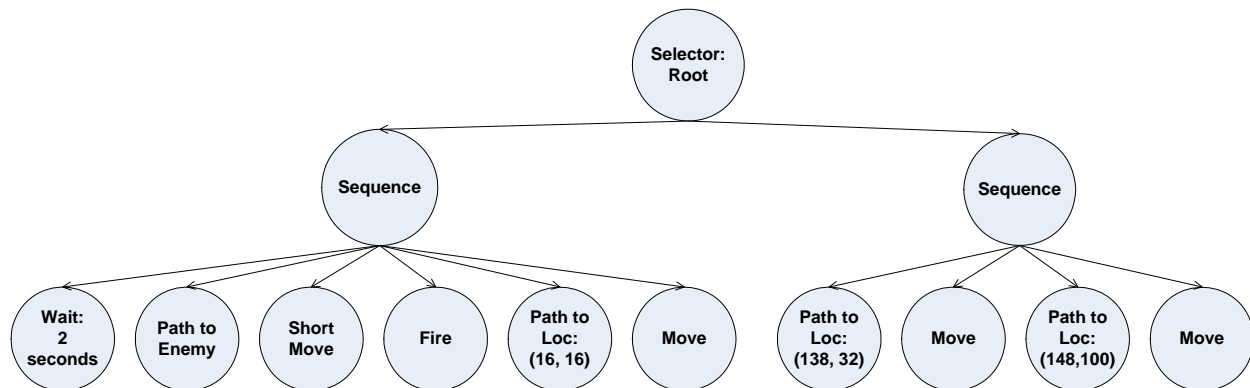
Behavior trees are implemented as a tree of objects extending the `BTNode` class. `BTNodes` have two primary methods, `Execute()` and `SetResult()`. `Execute()` is called each update cycle and causes the unit to perform some task. The result, a member of the `NodeResult` enumeration, signifies whether the task succeeded, failed, or is pending. If pending, the task needs to be executed again next update to see if its termination condition is met. Otherwise, the result is passed to the node's parent using `SetResult()`. In the case of the root node, no action is taken. Each `BTNode` represents a behavior or task, including:

- `FireBehavior`

- Causes the unit to damage its current target if the target is in range.
 - Fails if the target is out of range.
- MoveBehavior
 - Causes the unit to move along its currently active path.
 - Succeeds when destination is reached.
- Path2Enemy
 - Calculates and records the path from the unit's current position to a random enemy.
 - Fails if no path can be found.
- Path2Loc
 - Calculates and records the path from the unit's current position to a fixed location.
 - Fails if no path can be found.
- Selector
 - Executes each of its child nodes in order, until one succeeds.
 - If all children fail, this node fails.
- Sequence
 - Executes each of its child nodes in order, until one fails.
 - If all children succeed, this node succeeds.
- ShortMoveBehavior
 - Causes the unit to move along its currently active path, until its current target appears in range.
 - Succeeds when destination is reached, or target is in range.
- WaitBehavior
 - Causes the unit to wait for a specified unit of time.

BTAI is the unit class that uses behavior trees. It contains a reference to the root node in its behavior tree, and a node stack representing the path to the current node. This node stack allows the agent to quickly backtrack and go down a different branch of a tree when the current node terminates. Only the node on top of the stack is executed at during an update cycle. If the top node terminates (returns a `NodeResult` other than `PENDING`) and it is not the root, it is popped off the stack. Aggregate nodes (sequence and selector nodes) can push new nodes onto the node stack when they execute.

Behavior trees are encoded in XML; the specific one tested is mirrored in the structure shown below:



This tree can be thought of as two alternate behaviors, represented by the two sequence nodes. The sequence on the left represents a simple hit-and-run behavior, while the one on the right causes the unit to visit two specific points. If for some reason, a step on the left sub-tree fails, the failure will propagate up to the root node, which will switch to the behavior on the right. For example, if, while executing the “Short Move” behavior, the unit reaches its destination without ever seeing its target, the following “Fire” behavior will fail; The unit will then pace between points (138, 32) and (148, 100) before starting over.

Research Conclusion

The neural network will suffice as a replacement for a complicated rules based system. With the information gathered from this prototype we can see that the weaknesses will be a real time adaptation of the network.

Training the system manually will only get half of the desired results. The rest of the results will come from a heuristic evaluation of the performance of the neural network. This will allow the network to adapt to the players behavior and provide an interesting and engaging AI. The done in this prototype prove that this is possible and the speed of the network is fast enough to handle this kind of adaptation.

The network learned a training set of 200 in under 300 iterations. This equated to a time of just less than 1 second. The runtime performance of the network is fast enough to process the state of the game in less than a 1/60 of a second however this was not the primary concern for this test.

As far as the behavior tree research goes, it is a promising technology to use for the capstone. Complex behaviors can be written and tested quickly without recompiling the game code. No noticeable slowdown occurred during testing due to the execution of the behavior tree. There was a slowdown in relation to the A* algorithm used for pathfinding, but there are known optimizations that can be applied to solve this problem.

Overall, the combined neural net/behavior tree architecture shows promise, and should be an excellent foundation for the AI in the capstone game.

User Interface Technology Test

Testing the basic performance and functionality of the User Interface system.

Goals

The user interface technology test aims to implement and develop the framework described in the personal research of Colin Doody into user interface technologies and create a solution that works in the context of the game engine described for this game.

Requirements

In order to test the GUI system, a program will be established that implements a basic skeleton that reflects the proposed user interface system. This system does not need to implement all of the functionality that the final GUI system proposes, but instead should test some of the processes and procedures that can only be fully understood and proven through implementation.

Specifically, the User Interface Technology test should aim to reveal information on the following:

- **Will the sizing and transformation of GUI elements be “pixel perfect”?** The GUI system attempts to size and position elements by the pixel, and operate on these elements in a conceptual pixel space, no matter the resolution of the screen in the end. When rendered to the screen, however, the GUI elements will end up being defined in screen space, which operates on values between 0 and 1. Will the accuracy of the matrix manipulation result in crisp, pixel perfect positioning, after all transformations are taken into account? **In order to pass the test, GUI elements will need to demonstrate that they can be sized and positioned to the pixel according to their description at multiple resolutions.**
- **Will parenting yield the appropriate results?** Something that has not been attempted in previous implementations of the GUI interface structure are in regards to the parenting of GUI elements. It is important to make sure that all transformations occur appropriately, given that the GUI system operates in pixel space and takes extra transformations into account, such as a Stage scale. **In order to pass the test, GUI elements will need to demonstrate parenting and also behave appropriately under multiple transformative situations.**
- **Will the performance of the system be appropriate?** The system being implemented suggests optimizing speed by using a single vertex buffer maintained by the GUI core that gets sent to the card. All elements pass their properties to the card and manipulate the single vertex buffer. Will this method actually perform appropriately? **In order to pass the test, 400 GUI elements* should be able to render to the scene every frame and take less than a quarter of a single frame’s time (0.004166 seconds).**

** 400 GUI elements was an arbitrarily selected number of GUI elements that should accommodate most systems. Most systems will not require more than 400 GUI elements to be on the screen at a time, and should more elements need to exist, a developer is able to, at an application level, optimize their implementation to improve performance.*

Implementation

A skeleton GUI system was implemented to test the various questions outlined. A GUI core was implemented that is called to update and render the system. A scene was implemented, and a scene was updated and rendered as it would be in a final system. Finally, GUI elements were put into place with all of the properties that they would exhibit in the final system. All rendering took place within the shader program and used the single vertex buffer as proposed in the research.

Two implementations of the system have been put in place in order to test the three open questions with the GUI system.

The first implementation places GUI elements next to each other, side by side, separated by a single pixel. This test will not only be able to test the “pixel perfect” nature of the GUI system, but will also be used as a test of 400 GUI elements. Each element is set up to be 20 pixels wide, and is positioned 1 pixel apart from its neighboring elements.

The second implementation of the system creates a GUI element 300 pixels in size. This element is then given a scale of 0.5 in both the X and Y direction. Its resulting size is 150 pixels. A second GUI element is created and parented to the original size. The original parent is rotated. This tests the hierarchical transformation of elements by ensuring that a child element inherits the scale, relative offset and rotational parameters of its parent.

Results

The first test, determining whether or not image elements would be pixel perfect, resulted **positively**. Each element was exactly 20 pixels in width, as defined, and had one very clear pixel between them. This was also maintained in multiple resolutions of the window. When the window was established at 700 pixels by 450 pixels, and the stage was built with the same theoretical resolution, each element was the same exact size as elements that were built on a 500 pixel by 500 pixel window with a 500 pixel by 500 pixel stage. They all centered appropriately as well.

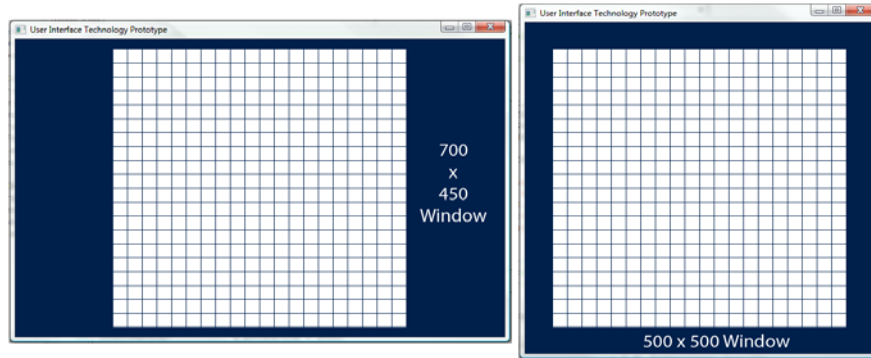


Figure 1: Two windows, one at 700 x 450 and the other at 500 x 500.

The second test, regarding parenting of systems, resulted **positively**. The child element ended up being exactly 10 pixels in size, half of what it should have been. Rotation occurred appropriately as well on the elements.

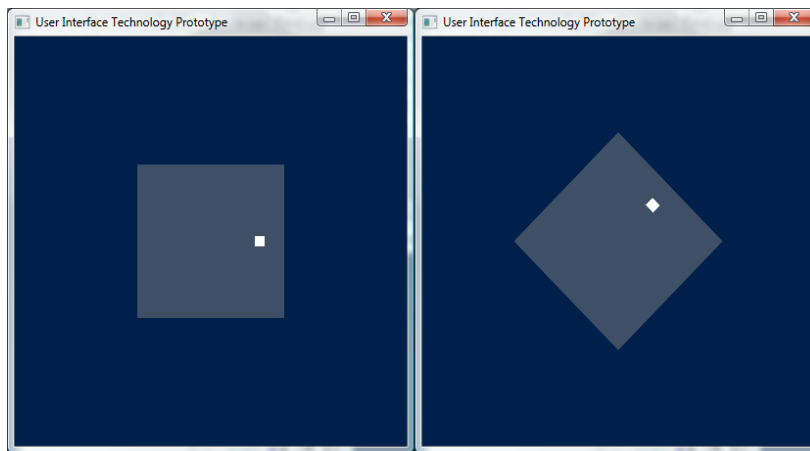


Figure 2: The results of the parenting test

The final test, a test of performance, tested **negatively**. 400 elements, stressed to update on every frame, yielded an average over 0.006477 seconds to a frame (out of three samples with the results of 0.006597, 0.006377, and 0.006477.) This missed the target speed of 0.004166 seconds to a frame.

Research Conclusion

Many of the open questions regarding the GUI system yielded positive results. Generally, the architecture proved fit to position objects, update objects and parent objects. Unfortunately, performance became an issue in the system. Future study of performance will need to be addressed. It is possible that the way the minimal testing platform was set up is affecting the performance of the system. It is also more likely that functions implemented within the system need to be reviewed and tweaked to improve performance.

Extra Notes

The technology demo that went along with this paper can be found in the `UserInterface` folder of the `demos` directory.

Curiosity Studios

Working Title: Tank Frenzy

Cooperative, Tactical Tank Combat in a Living Battle

Appendix

Matthew Bozarth - Colin Doody - Andrew Galante - Joshua Gilpatrick - Nick Korn
2/27/2009

Appendix A: Assets List

Assets required to develop the game

Mesh Assets

Tanks

NAME	FILENAME	Description
Hornet (Body)	MESH_BODY_Hornet	
Hornet (Turret)	MESH_TURRET_Hornet	
Hornet (Machine Gun)	MESH_MG_Hornet	
Rhino (Body)	MESH_BODY_Rhino	
Rhino (Turret)	MESH_TURRET_Rhino	
Rhino (Machine Gun)	MESH_MG_Rhino	
Lancer (Body)	MESH_BODY_Lancer	
Lancer (Turret)	MESH_TURRET_Lancer	
Lancer (Machine Gun)	MESH_MG_Lancer	
Ant (Body)	MESH_BODY_Ant	
Ant (Turret)	MESH_TURRET_Ant	
Ant (Machine Gun)	MESH_MG_Ant	
Light Tank (Body)	MESH_BODY_LightTank	
Light (Turret)	MESH_TURRET_LightTank	

Battle Tank (Body)	MESH_BODY_BattleTank
Battle Tank (Turret)	MESH_TURRET_BattleTank
Artillery (Body)	MESH_BODY_Artillery
Artillery (Turret)	MESH_TURRET_Artillery
Heavy Tank (Body)	MESH_BODY_HeavyTank
Heavy Tank (Turret)	MESH_TURRET_HeavyTank
Turret (Body)	MESH_BODY_Turret
Turret (Turret)	MESH_TURRET_Turret

Environment

NAME	FILENAME	DESCRIPTION
Tree	MESH_Tree	
Rock	MESH_Rock	
Boulder	MESH_Boulder	
Building 1-20	MESH_Building_01- MESH_Building_20	
Bridge	MESH_Bridge	
Wall Segment	MESH_Wall	

Wall Joint

MESH_WallJoint

Texture Assets

Tanks

NAME	FILENAME	DESCRIPTION
Hornet (Body)	TEXTURE_BODY_Hornet	
Hornet (Turret)	TEXTURE_TURRET_Hornet	
Hornet (Machine Gun)	TEXTURE_MG_Hornet	
Rhino (Body)	TEXTURE_BODY_Rhino	
Rhino (Turret)	TEXTURE_TURRET_Rhino	
Rhino (Machine Gun)	TEXTURE_MG_Rhino	
Lancer (Body)	TEXTURE_BODY_Lancer	
Lancer (Turret)	TEXTURE_TURRET_Lancer	
Lancer (Machine Gun)	TEXTURE_MG_Lancer	
Ant (Body)	TEXTURE_BODY_Ant	
Ant (Turret)	TEXTURE_TURRET_Ant	

Ant (Machine Gun)	TEXTURE_MG_Ant
Light Tank (Body)	TEXTURE_BODY_LightTank
Light (Turret)	TEXTURE_TURRET_LightTank
Battle Tank (Body)	TEXTURE_BODY_BattleTank
Battle Tank (Turret)	TEXTURE_TURRET_BattleTank
Artillery (Body)	TEXTURE_BODY_Artillery
Artillery (Turret)	TEXTURE_TURRET_Artillery
Heavy Tank (Body)	TEXTURE_BODY_HeavyTank
Heavy Tank (Turret)	TEXTURE_TURRET_HeavyTank
Turret (Body)	TEXTURE_BODY_Turret
Turret (Turret)	TEXTURE_TURRET_Turret

Environment

NAME	FILENAME	DESCRIPTION
Tree	TEXTURE_Tree	

Rock	TEXTURE_Rock
Boulder	TEXTURE_Boulder
Building 1-20	TEXTURE_Building_01- MESH_Building_20
Bridge	TEXTURE_Bridge
Wall Segment	TEXTURE_Wall
Wall Joint	TEXTURE_WallJoint
Terrain Textures	TEXTURE_Terrain01- TEXTURE_TerrainXX

Particles

NAME	FILENAME	DESCRIPTION
Explosion Particle	TEXTURE_PARTICLE_Explosion	
Spark Particle	TEXTURE_PARTICLE_Spark	
Water Particle	TEXTURE_PARTICLE_Water	
Snow Particle	TEXTURE_PARTICLE_Snow	

Gun Fire Particle	TEXTURE_PARTICLE_GunFire
Smoke Particle	TEXTURE_PARTICLE_Smoke
Shield Hit	TEXTURE_PARTICLE_Shield

Level Files

NAME	FILENAME	DESCRIPTION
Fort Ashford	LEVEL_FortAshford	

Particle Effect Descriptions

NAME	FILENAME	DESCRIPTION
Explosion	PARTICLE_DESC_Explosion	
Spark	PARTICLE_DESC_Spark	
Water Explosion	PARTICLE_DESC_WaterExplosion	
Snow Explosion	PARTICLE_DESC_SnowExplosion	
Gun Fire	PARTICLE_DESC_GunFire	
Smoke	PARTICLE_DESC_Smoke	
Snow	PARTICLE_DESC_Snow	
Shield Hit	PARTICLE_DESC_ShieldHit	

Audio Assets

User Interface Sound Effects

NAME	FILENAME	STATES	DESCRIPTION
Major Button Mouse Over	SFX_UI_MajorBtnMouseOver		
Major Button Mouse Out	SFX_UI_MajorBtnMouseOut		
Minor Button Mouse Over	SFX_UI_MinorBtnMouseOver		
Minor Button Mouse Out	SFX_UI_MinorBtnMouseOut		
State Move Forward	SFX_UI_StateForward		
State Move Back	SFX_UI_StateBack		
Keyboard Type	SFX_UI_KeyType		
Menu Item Highlighted	SFX_UI_MenuItemHighlight		
Menu Item Clicked	SFX_UI_MenuItemClicked		
Screen Back	SFX_UI_ScreenBack		
Typing	SFX_UI_Typing		
Connecting	SXF_UI_Connecting		

Action / World Sound Effects

NAME	FILENAME	DESCRIPTION
Heavy Cannon Fire	SFX_ACT_HeavyCannon	

Medium Cannon Fire	SFX_ACT_MediumCannon
Light Cannon Fire	SFX_ACT_LightCannon
Long Range Cannon Fire	SFX_ACT_LongCannon
Machine Gun Fire	SFX_ACT_MachineGun
Shell Metal Hit	SFX_ACT_ShellMetalHit
Shell Snow Hit	SFX_ACT_ShellSnowHit
Shell Water Hit	SFX_ACT_ShellWaterHit
Tank Roll	SFX_ACT_TankRoll
Machine Gun Metal Hit	SFX_ACT_MGMetalHit
Machine Gun Snow Hit	SFX_ACT_MGSnowHit
Machine Gun Water Hit	SFX_ACT_MGWaterHit
Player Explosion	SFX_ACT_PlayerExplode
Tank Explosion	SFX_ACT_TankExplosion
Building Explosion	SFX_ACT_BuildingExplosion

Environmental Sound Effects

NAME	FILENAME	MISSION	DESCRIPTION
Forest	SFX_ENV_Forest	General	
Tundra	SFX_ENV_Tundra	General	
Industrial	SFX_ENV_Industrial	Fort Ashford	
Oil Center	SFX_ENV_OilCenter	Fort Ashford	

Notification Sound Effects

NAME	FILENAME	DESCRIPTION
Game Paused	SFX_NTIFY_Pause	
Player Death	SFX_NTIFY_PlayerDeath	
Objective Complete	SFX_NTIFY_ObjectiveComplete	
Mission Complete	SFX_NTIFY_MissionComplete	
Server Disconnect	SFX_NTF_Disconnect	
Game Resumed	SFX_NTIFY_Resume	
Respawn	SFX_NTIFY_Respawn	
Teammate Lost	SFX_NTIFY_TeamDeath	
Teammate Respawn	SFX_NTIFY_TeamSpawn	

Dialogue Sound Clips

NAME	FILENAME	MISSION	DESCRIPTION
Mission Complete	SFX_DIALG_MissionComplete	General	
Objective Complete	SFX_DIALG_MissionComplete	General	
Mission Lost (Death)	SFX_DIALG_MissionFailDeath	General	
Mission Lost (Objective)	SFX_DIALG_MissionFailObjective	General	
Fort Ashford Intro	SFX_DIALG_FortAshfordIntro	Fort Ashford	
Fort Ashford Event I	SFX_DIALG_FortAshfordEvtI	Fort Ashford	
Fort Ashford Event II	SFX_DIALG_FortAshfordEvtII	Fort Ashford	

Music

NAME	FILENAME	MISSION	DESCRIPTION
Menu Music	MUSIC_Menu	General	
Fort Ashford Music	MUSIC_FortAshford		
Victory Music	MUSIC_Victory		
Defeat Music	MUSIC_Defeat		

Appendix B: Third Party Requirements

Third Party Requirements

Purpose

This document provides a list of all the external tools, assets and libraries we will be using in our project.

Tools

Title: Grome

Version: 1.2

Company: Quad Software

Description: Terrain creation utility which will be used as a level editor for our game. It allows for the creation of terrain as well as the placement of objects.

Website: <http://www.quadsoftware.com/>

Title: Maya

Version: 2008

Company: Autodesk Inc.

Description: 3D modeling software

Website: <http://www.autodesk.com/>

Title: Photoshop

Version: CS3

Company: Adobe Systems Inc.

Description: Image editing software.

Website: <http://www.adobe.com/>

Title: Fruity Loops Studio

Version: 8.0

Company: Image Line

Description: Digital audio workstation

Website: <http://flstudio.image-line.com/>

Title: Audacity

Version: 1.2.6

Company: Open source

Description: Open source software for recording and editing sounds.

Website: <http://audacity.sourceforge.net/>

Title: Pro Tools

Version: 8.0

Company: Digidesign

Description: Sound creation and production system

Website: <http://www.digidesign.com/>

Title: VTune™ Analyzers

Version: 9.1

Company: Intel Corporation

Description: C++ performance tools which include a thread checker as well as thread profiling.

Website: <http://www.intel.com>

Assets

Title: Sci-Fi Tanks Collection

Creator: 3DRT

Description: A collection of 11 sci-fi tank models which will be used for allied and enemy tanks in our game.

Website: http://www.3drt.com/3dm/sf-tanks/sci-fi_tanks-shots.htm

Licensing Information: <http://www.3drt.com/3dm/license.htm>

Title: Industrial Kit

Creator: 3DRT

Description: A collection of building, containers, and cranes.

Website: http://www.3drt.com/3dm/levels/industrial_set/industrial_set.htm

Licensing Information: <http://www.3drt.com/3dm/license.htm>

Title: Urban Construction Kit

Creator: 3DRT

Description: A collection of building in various states: modern, night lights, destroyed

Website: http://www.3drt.com/3dm/levels/urban_set/urban_set.htm

Licensing Information: <http://www.3drt.com/3dm/license.htm>

Libraries

Title: DirectX

Version: 10.0

Description: A graphics API designed for the Windows Vista platform

Source/Website: <http://msdn.microsoft.com/en-us/directx/default.aspx>

Licensing Information: None provided.

Title: FMOD

Version: 4

Description: An audio engine for game developers.

Source/Website: <http://www.fmod.org/index.php/products/fmodex>

Licensing Information: <http://www.fmod.org/index.php/sales>

Title: Visual Leak Detector

Version: 1.9g (Beta)

Description: A free, robust, open-source memory leak detection system for Visual C++.

Source/Website: <http://dmoulding.googlepages.com/vld>

Licensing Information: None provided.

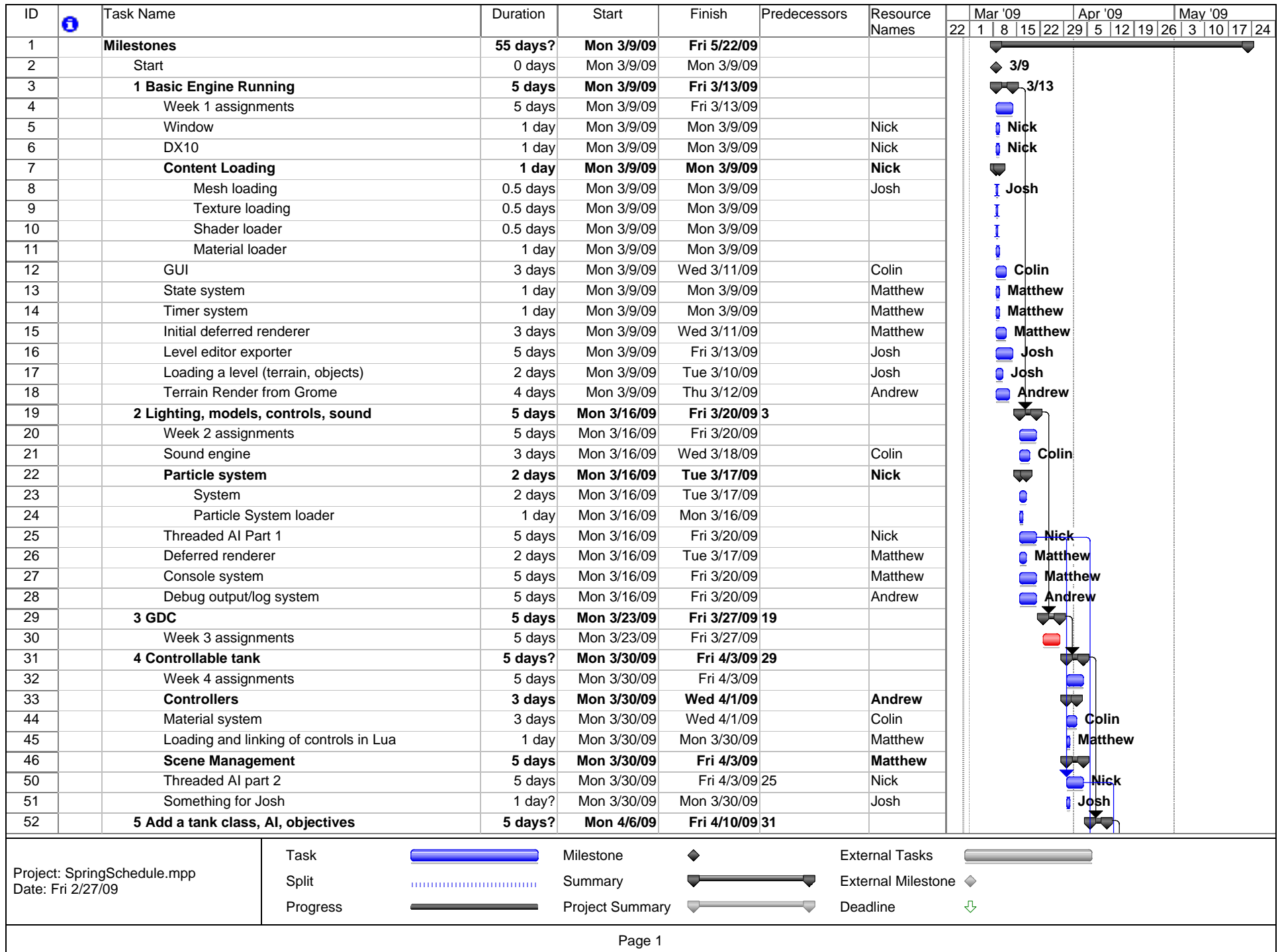
Appendix C: Scheduling Information

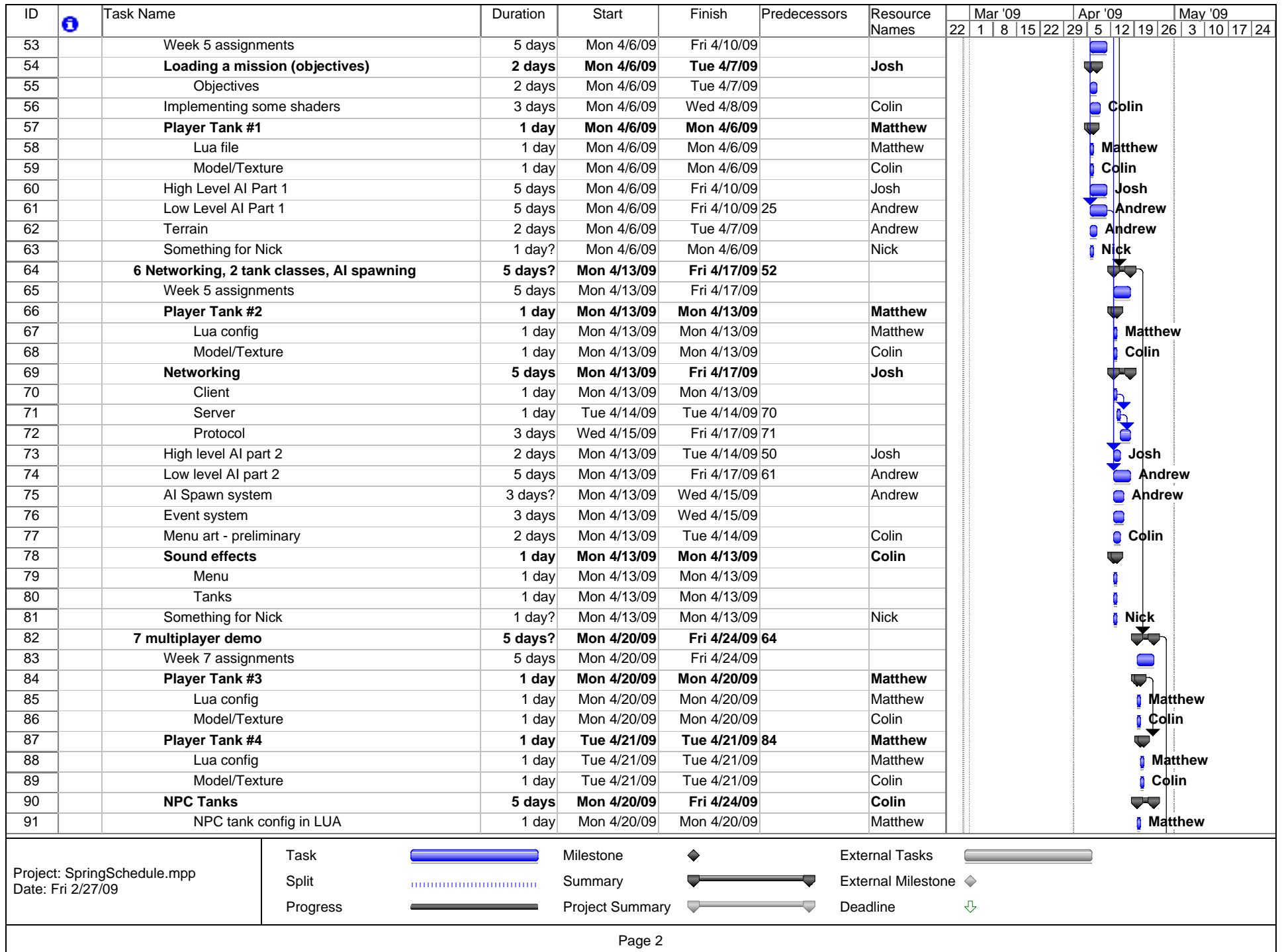
Scheduling Information for the development of the game

Chart I: Winter, 2009

Chart II: Spring, 2009

	Week 03		Break		Break		Week 04		Week 05		Week 06		Week 07		Week 08		Week 09		Week 10		Week 11							
	Dec 04 - Dec 10		Dec 11 - Dec 17		Dec 18 - Dec 24		Dec 25 - Jan 01		Jan 02 - Jan 08		Jan 09 - Jan 15		Jan 16 - Jan 22		Jan 23 - Feb 04		Feb 05 - Feb 11		Feb 12 - Feb 18		Feb 19 - Feb 25							
	S	M	T	W	R	F	S	S	M	T	W	R	F	S	S	M	T	W	R	F	S	S	M	T	W	R	F	S
1 Pitch Document																												
1.1 Elevator Pitch																												
1.2 High Concept Document																												
1.3 Game Treatment Document																												
1.4 Pitch																												
2 Game Design Document																												
2.1 Artificial Intelligence																												
2.2 Game Overview / Setting																												
2.3 World Overview																												
2.4 Player Classes / Abilities																												
2.5 Camera Settings / Movement																												
2.6 Controls																												
2.7 User Interface																												
2.8 Scripting																												
2.9 Networking																												
3 Level Design Document																												
3.1 Overview																												
3.2 Layout																												
3.3 Scripted Events																												
3.4 Objectives																												
4 Technical Design Document																												
4.1 Coding Guidelines																												
4.2 Engine																												
4.3 Scripting																												
4.4 Networking																												
4.5 Artificial Intelligence																												
4.6 Level																												
4.7 User Interface System																												
5 Art Bible																												
5.1 Trend Boards and Themes																												
5.2 2D Conceptual Art																												
5.3 User Interface Components																												
5.4 Conceptual Screenshots																												
5.5 Renders																												
6 Personal Assessment Document																												
6.1 Matthew Bozarth																												
6.1.1 Definition of Problem																												
6.1.2 Research into Problem Definition																												
6.1.3 Review of past implementations																												
6.1.4 Review of possible solutions																												
6.1.5 Solution and implementation																												
6.2 Colin Doody																												
6.2.1 Definition of Problem																												
6.2.2 Research into Problem Definition																												
6.2.3 Review of past implementations																												
6.2.4 Review of possible solutions																												
6.2.5 Solution and implementation																												
6.3 Andrew Galante																												
6.3.1 Definition of Problem																												
6.3.2 Research into Problem Definition																												
6.3.3 Review of past implementations																												
6.3.4 Review of possible solutions																												
6.3.5 Solution and implementation																												
6.4 Joshua Gilpatrick																												
6.4.1 Definition of Problem																												
6.4.2 Research into Problem Definition																												
6.4.3 Review of past implementations																												
6.4.4 Review of possible solutions																												
6.4.5 Solution and implementation																												
6.5 Nick Korn																												
6.5.1 Definition of Problem																												
6.5.2 Research into Problem Definition																												
6.5.3 Review of past implementations																												
6.5.4 Review of possible solutions																												
6.5.5 Solution and implementation																												
7 Technology Tests																												
7.1 Artificial Intelligence Prototype																												
7.2 User Interface Technology Test																												
7.3 Mesh File Formats Test																												
8 Working Prototype																												





ID	Task Name	Duration	Start	Finish	Predecessors	Resource Names	Mar '09					Apr '09				May '09				
							22	1	8	15	22	29	5	12	19	26	3	10	17	24
92	NPC tank models	5 days	Mon 4/20/09	Fri 4/24/09		Colin														
93	Menu art - final	2 days	Mon 4/20/09	Tue 4/21/09		Colin														
94	Something for Nick	1 day?	Mon 4/20/09	Mon 4/20/09		Nick														
95	Something for Andrew	1 day?	Mon 4/20/09	Mon 4/20/09		Andrew														
96	8 All classes, 4 player support	5 days?	Mon 4/27/09	Fri 5/1/09	82															
97	Week 8 assignments	5 days	Mon 4/27/09	Fri 5/1/09																
98	Mission: start to finish	0 days?	Fri 5/1/09	Fri 5/1/09																
99	Voice acting	1 day	Mon 4/27/09	Mon 4/27/09		Colin														
100	Music	4 days	Mon 4/27/09	Thu 4/30/09		Colin														
101	Level 1	4 days	Mon 4/27/09	Thu 4/30/09		Matthew														
102	Something for Nick	1 day?	Mon 4/27/09	Mon 4/27/09		Nick														
103	Something for Josh	1 day?	Mon 4/27/09	Mon 4/27/09		Josh														
104	Something for Andrew	1 day?	Mon 4/27/09	Mon 4/27/09		Andrew														
105	9 Alpha	5 days	Mon 5/4/09	Fri 5/8/09	96															
106	Week 9 assignments	5 days	Mon 5/4/09	Fri 5/8/09																
107	Level 2	4 days	Mon 5/4/09	Thu 5/7/09		Colin														
108	10 Beta	5 days	Mon 5/11/09	Fri 5/15/09	105															
109	Week 10 assignments	5 days	Mon 5/11/09	Fri 5/15/09																
110	11 Gold	5 days	Mon 5/18/09	Fri 5/22/09	108															
111	Week 11 Assignments	5 days	Mon 5/18/09	Fri 5/22/09																



Project: SpringSchedule.mpp
Date: Fri 2/27/09

Task



Milestone



External Tasks



Split



Summary



External Milestone



Progress



Project Summary



Deadline

