Project Number: RL1-ALI1

The Alligator: A Video Game History of a Civil War Submarine

A Major Qualifying Project Report:
submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

_____

Dana Asplund

_____

Kalun Fu

_____

Yilmaz Kiymaz

_____

Timothy Loughlin

Date: January 7, 2007
Approved:

_____

Professor Robert W Lindeman, Major Advisor

_____

Chuck Veit, Sponsor

Abstract

Alligator, a three-dimensional Civil War submarine simulator, is designed to teach middle and high school students the basic concepts of naval navigation and tactics. Developed for the Navy and Marine Living History Association, the project attempts to realistically portray the hardships and technological challenges of piloting the Union Navy's first submarine. Students will have to plan attack and escape routes, compensate for tides and currents, and execute their plan in a full three-dimensional environment.

By encouraging group participation and debate through the Mission Planning stage, Alligator provides an interesting classroom environment for learning about the Civil War and naval tactics. While the execution gameplay stage of the game is focused upon single player action, the Mission Planning stage is designed to be used by groups of students. Each group plans their own route and the class as a whole then analyzes and debates the individual groups' plans before choosing the most appropriate course of action to execute.

The historical accuracy of the simulation is critical to the design of the project, for the players must gain a sense of the difficulties in handling the Alligator. The physical and psychological struggles of the crew, which greatly affect their survival and the success of the mission, are made clear throughout the game. The available technologies and tools during this time period are limited, and so the students must deal with these issues.

We aimed to produce a software application that is both enjoyable and educational for the player. With the conclusion of the project, we have developed an impressive prototype that will be extended through future development, by other students or professionals.

Acknowledgements

We would like to extend our thanks to Chuck Veit, our sponsor, for contributing invaluable insight, references, models, and encouragement during the project. Without his active involvement and support, we would never have been able to deliver such an interesting and historically accurate project.

We would also like to thank Robert Lindeman for advising this project, and for his direction and attention to detail. We thank him for supplying a vast amount of Artificial Intelligence material, which greatly aided our understanding of AI concepts and implementation.

Our thanks also go out to the C4 engine forums community at http://www.terathon.com. Their rapid responses and helpful answers to our many issues helped us achieve as much as we did during the project's time span.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The Alligator: A Video Game History of a Civil War Submarine was focused upon the development of a complex and historically accurate simulation of the Alligator submarine and its proposed missions. The game's purpose is to teach naval concepts such as navigation, submarine maneuvering, and submarine combat tactics. The game requirements included both a 2D mission planning mode, and a 3D gameplay mode.

The completion of such a project requires proper object-oriented design and analysis, as well as consistent and realistic project scheduling and management. Due to the complexities of producing a professional quality 3D game, we aimed to produce a viable prototype for the sponsor, the Navy and Marine Living History Association. The prototype will be used for obtaining feedback about the game, and securing funding for continued development of the game for full release.

We designed the art assets, physics, weapons, and technologies to be as historically accurate as possible. Reference photographs, schematic diagrams, and period writings were used in modeling the vessels, weapons, and characters in the game. The terrain was generated from actual satellite maps of the historic mission locations, including Norfolk, Virginia and the nearby Hampton Roads. We strove to implement realistic physics for the 3D gameplay as well, and developed rigid body kinematics, tides, water current forces, buoyancy, and fluid viscosity and drag. Environment and weather conditions were also simulated, with real-time day and night cycles, clouds, fog, rain, and snow.

This report details the full design of the game, much of which was beyond the scope of the semester-long project. The information in the design section is included regardless of current implementation, for the purposes of future development. We also detail specifics about the project, such as what was successful, what proved problematic, and what we would have done differently.

2 Game Design

2.1 Game Description

The Civil War's first submarine, the Alligator, is back from the depths of the sea to deal out damage to the Confederate Navy. With the invention of the Alligator by Frenchman Brutus de Villeroi, the Union got the upper-hand in naval combat [1]. Plan out your strategies to destroy or recover your targets using accurate and detailed maps of the Confederate's defensive position during the 1800s, depth of water for the rivers and waterways, time of day, and weather in order to maximize chance of success. Execute your plan by maneuvering the Alligator through riverbeds scattered with sunken ships, rock beds, and underwater mines while avoiding enemy ship patrols and blockades, defensive gun batteries, scout towers equipped with spotlights, and watchmen. With the cover of night and mastery over the Alligator and its crew you will be able to sink any ship, destroy any blockade, remove any underwater obstacles, and escape any enemy attack. The real Alligator has never seen the heat of battle, and so in this historically accurate simulation you will relive history in a way that has never been experienced before.

2.2 Overview

Alligator is a submarine simulation unlike any other. Unlike the modern-day simulators, Alligator is set in the time of the Civil War, the late 1800s, with no GPS systems, high-tech gadgets and screens, guided torpedoes, or computerized navigational systems. The players use the information given during Mission Planning to design their strategy and navigational routes based on tides and currents, enemy locations, water depths, underwater obstructions, and the like. The players work out their routes by hand using methods and techniques of nautical navigation, and execute their plans using limited tools such as candle-lit maps, visual sightings, and needle pressure gauges. The crew experiences fatigue, making navigation harder, and may even suffer from asphyxiation from the undetectable carbon dioxide levels in the submarine. The players also have to use their own awareness to navigate by determining their position and orientation based

on landmarks seen from the conning tower. Mostly, though, the students have to be able to follow their plans of navigation to accommodate tides and currents so that they aren't tossed into obstructions, such as mines, that lead to disaster.

2.3 Target Audience

Alligator is designed to be played by students in a high-school classroom or in a museum. The game is meant to teach students about the idiosyncrasies of early submarine pioneering when it comes to naval combat, navigation, and submarine management. Therefore, the game is meant to be played in a single class period or spread out over relatively few class periods, estimating a class period to be under an hour.

2.4 Target Platform

The project is compiled for Windows XP. Since the project uses the C4 game engine, it can be ported to Macintosh in the future, as well as to the Playstation 3 [4].

2.5 Gameplay

The game consists of two main parts, two-dimensional mission planning and three-dimensional execution.

2.5.1 Alligator Modes

The Alligator submarine was designed with two propulsion modes and two attack modes.

For propulsion, the Alligator could be fitted with oars along both sides of the vessel, which crewmen would operate to propel the vessel. Alternatively, the propeller mode fitted a screw propeller, which crewmen would hand-crank to operate [1].

The main mode of attack for the submarine was the diver, who would be released from the forward compartment with explosives. He would have to walk along the river or ocean floor, and place the explosives within range of the target. The optional method of attack was the man-hole attack, where the Conning Tower would be removed and replaced with the top half of a diver suit. This allowed the captain to dictate movement to

the crew, bringing the submarine to within a few feet of the target, and place the explosives himself directly without a diver [1].

We have implemented the oar movement and 3D model, and the diver attack mode.

2.5.2 Mission Planning

Mission Planning is the first stage of the game where players will have to review historical information and plan out the mission strategies they will use. Mission planning has a tabbed structure to present all the mission information to the players, and to allow the player to plan out his attack. It also has an interactive map to display a visual area of the mission, allowing the players to plot their attack and return routes.

The Mission Planning section of the game is where the students should be involved as a class. The class will most likely be broken up into teams, and each team will have to choose a different route to plan for. They will have to come up with the best route based on all the variables of the path and determine things such as total length and time for the path, reorientations and speeds for the route (taking into account the tides and currents), mid-mission goals to accomplish, and other factors. The players will plot their desired paths by placing waypoints on the map. The map will be able to determine the distance and orientation between waypoints and the exact water depth, tide, and current at each waypoint [2].

Once each team has their plan formulated, the entire class will present and explain their plans and estimated success rates of it. The class will analyze all the plans and determine the best one to execute. Each student (or team) will execute the chosen plan (or each student will choose a plan they want to execute and execute it. This will make it possible to prove which plan *was* actually the best). This way, many routes can be covered by the class and many routes can be executed to determine which plan was best, and therefore analyze the data (success of the mission) to determine reasons for success or failure (maybe the team who made the best plan and the team which best executed a plan get bonus points in the class).

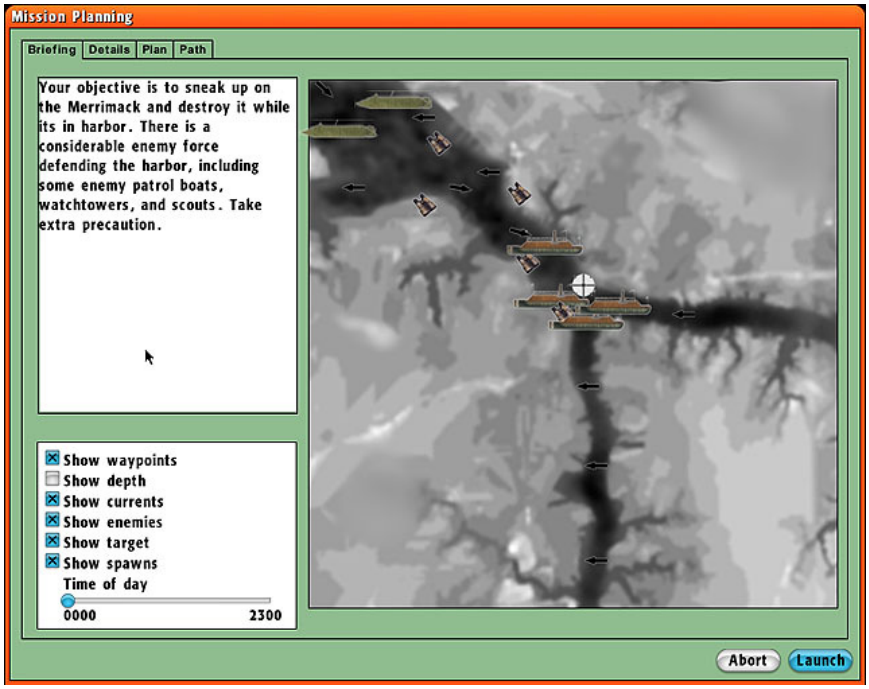

Figure 1 - Mission Planning Map and Tabbed Structure

The most important aspect of Mission Planning is the map, shown in Figure 1. This has icons or symbols on it to show the players the following: enemy ships, gun battery defenses, scout towers, landmarks, underwater obstructions, water depth grades, and current directions. The players are able to see the different views or overlays for each of these pieces of information, that is, to make visible or transparent the location of enemies, underwater obstructions, and current directions in order to see isolated components of the mission on the map.

The first tab is the Briefing tab, which provides a description of the target and the required goals. This may include the best ways to approach the target and handle accomplishing the mission, such as describing how alert and watchful the enemies are.

The next tab, Details, tells the players which areas of intelligence are especially significant to the mission, such as where intelligence could not survey for defenses or enemy positions, underwater mines, obstructions, etc. For instance, the mission may be to destroy the *Merrimack* while she is in port, so the Details will detail the fact that the submerged section of the hull of the ship is susceptible to an explosion within 10-15 feet.

The Details may state that the ship is heavily guarded at night but rather empty during the day, as well as provide threat levels for certain areas.

Either of these tabs may be used to describe the historical or tactical reasons for this mission, the generals and leaders of the mission, and what the success or failure of this mission means in the grand scheme of the war. This will give some background of the Civil War at the time of this mission and what was occurring for the Union or Confederate Army/Navy at the time. A more elegant solution would be to play pre-rendered video clips or photo montages, both possible through the C4 engine's Quicktime movie plug-in [4].

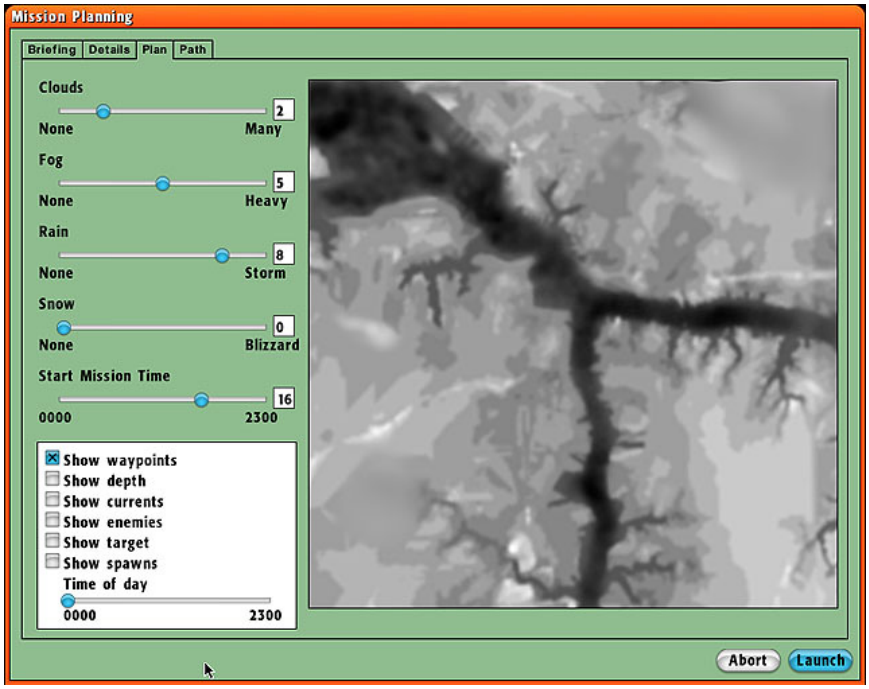

Figure 2 - Mission Planning Weather and Time Settings

Mission Planning also contains a tab labeled Plan, as shown in Figure 2, allowing the players to design their environmental conditions for accomplishing the mission. The players may choose their desired mission settings through the use of the following sliders: clouds, fog, rain (high level of rain yield lightning and thunder), snow, and time of day. Water visibility in terms of mud, algae, and others are mission-based conditions and cannot be set. Depending on the time of day chosen the tides will be different, either high

tide or low tide, which will change the depths of the waters, thereby changing some possible attack routes [2]. The requirements for this feature are provided, but the tides themselves are not currently implemented. A function to adjust the water level dynamically according to time (as the sky and lights do) may be written in the future, as well as one to change the direction of the current forces.

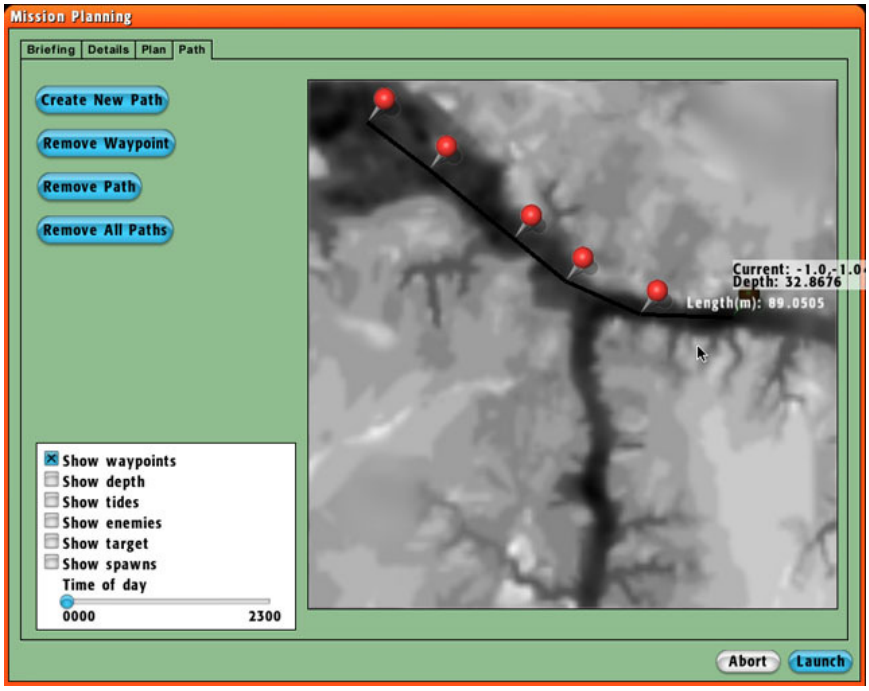

Figure 3 - Mission Planning Paths and Waypoints

The final tab in Mission planning, the Path tab, shown in Figure 3, is where players can create new paths, remove waypoints from paths, remove an entire path, and remove all paths. Waypoints are placed with mouse clicks on the map itself, and may be placed in any of the tab windows. Once placed, clicking on a waypoint icon will select it, and display its information. The Path tab provides higher level control for planning an entry and exit path, and deletion of waypoints and paths.

Several features of Mission Planning we were not able to include are of note for future development. The players should be able to set the number of divers they are sending along (max of 2), as well as how many explosives. Selection of the man-hole attack mode or diver attack mode may be included (the man-hole mode has not been

implemented), or may be mission based. The player should also be able to choose his Spawn Point, either from a given number of preset locations designated by the level designer, or any location out of range of enemy sight of the Alligator. The players may be given suggested paths to follow in order to accomplish the mission. The players should also plan an attack route and a return route, probably changing the return route to escape enemy retaliation (such as leaving in a shallow area where ships could not follow). The player's attack and return paths may be shown in red and blue, or other distinctive colors. While the interface currently allows for multiple path creation, colors are not changed.

2.5.3 Mission Execution

The game starts with a 1st person view from the Conning Tower of the submarine, as shown in Figure 4. For future development, a scripted or animated 3rd person view of the Alligator being towed to the chosen Release Point would be more appealing. Upon release, the camera could animate into the 1st person view of the Conning Tower or submarine interior.

Figure 4 - Conning Tower View and Button Panel

The player then has to navigate using both the in-game map and compass, and the button control panel. He has to observe his surroundings for landmarks and features to determine if the sub's direction and location are correct. The player has to navigate the sub using the rudder, and crew commands to surface, dive, and maneuver the sub. The sub only needs to travel several miles, at a top speed of around five to six knots (around 6.9mph) [3]. To travel two miles, it would take roughly twenty minutes to go at full speed the entire way. The player will have to consider the currents and tides and accommodate accordingly.

Another thing to consider is the air circulation. The player must be aware of what is happening with the air to make sure there is enough air for the crew at all times. When oxygen levels are low, the crew will complain of discomfort and sickness. The player must decide whether or not to surface, so the air hose may draw air in from the outside. If surrounded by enemy vessels and defenses, this would be difficult.

Figure 5 - Captain's Station View

The player can switch the camera placement inside the sub, changing the view that is displayed. The current game has a Change Views button, which cycles through the

Conning Tower view, Captain's Station view as shown in Figure 5, map view, and the compass view. If the Captain has deployed the diver, then the player takes the view of the diver and controls the diver. This is only for diver attack mode; if the sub is in man-hole attack mode then he will not control the diver but will only signal to place the explosive on the target, maybe with an animation or cinematic of the placement. The man-hole attack mode has not been implemented in this version of the game.

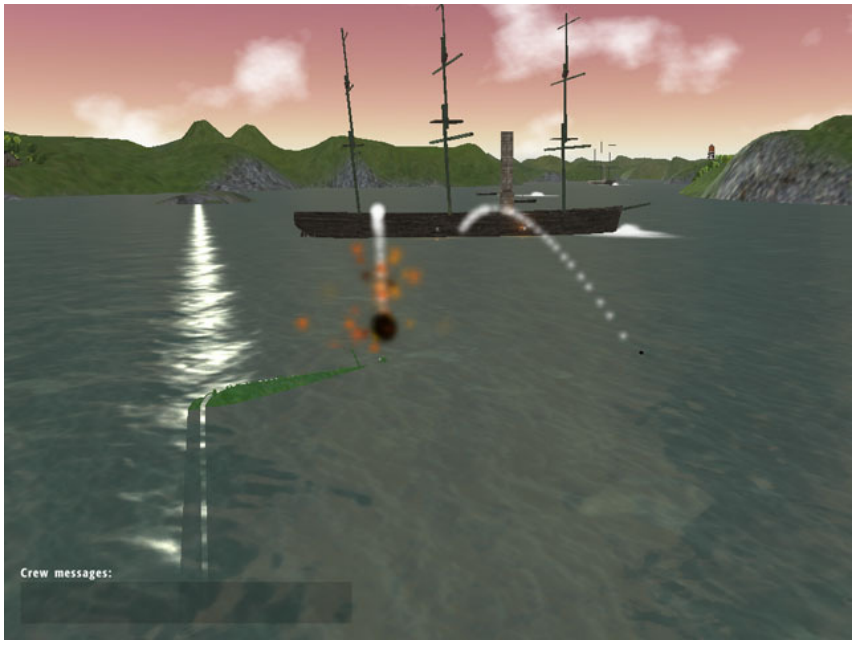

Figure 6 - Enemy Frigate Firing on the Alligator

The enemy units can spot the submarine and attack, as shown in Figure 6. Currently, frigates are implemented, with five cannons on each side. These ships may locate the Alligator (depending on weather, time, and depth), and will maneuver and fire upon the submarine.

Figure 7 - The Diver Armed With Explosives

Once the sub is in place for an attack, the Captain gives the order to deploy the diver. Once deployed, control and camera view are from the diver's perspective, as shown in Figure 7. For future development, an animation of the diver being sealed in his chamber, changing into his suit, and the chamber filling with water may be added. The diver then travels to the target, places explosives, reaches a safe distance, and detonates the explosives. If the objective is destroyed, the game is won.

In the future, several improvements to the gameplay may be added. The diver should return to the submarine, giving control and camera back to the Captain, and allowing the Alligator to move out of range of the explosion before detonation. Once the explosives are triggered, the player must retreat from the area safely before the mission is finished. We have added two other enemy units, Picket Boats and Gun Batteries. These require further development before they are fully functional. Other units may be added, as well as other weapons such as muskets and chain nets to drop on the diver.

3 Game World

3.1 Characters

The main characters for the game are the Captain, the diver, and the crew. There are enemy ships and gun batteries, and more may be added such as humans and guard towers. The players will generally control the Captain when maneuvering the Alligator and the diver when making the attack on the target.

3.2 Entities

Entities in the C4 engine are animated, moveable models. They are used for anything that requires animation, but are essentially geometry models with animation files attached. Controllers in the C4 engine are classes that are automatically called every frame render, allowing the class to update any other object during runtime. Entities in C4 are commonly paired with a controller to set the current animation, provide input and movement, and other updateable features.

The currently implemented vehicle entities in the game include the following: the Alligator submarine, frigates, ironclads, picket boats, and gun batteries [3]. Other entities include: rowers, divers, and the compass. For future development, other entities may be added such as rowboats, riflemen, bridges, towers, and biological entities such as fish and sharks.

Weapons are also designed as entities. The game features the following weapons: explosives, cannons, and mines. In the future, rifles and chain nets should be added.

3.3 Project Missions

The following two missions are the currently implemented levels for the game. We have also included additional mission scenarios in the Post-Project Missions section.

3.3.1 Sink the Merrimack (CSS Virginia), Norfolk, VA

Description: "Initially, it was hoped that the submarine would be able to sink the *Merrimack*. Whether this would actually have been possible is a matter of debate. Certainly the diver would have had little chance of success were the rebel ironclad underway; *Alligator* would have had to attack while the enemy vessel was in dock. Its ability to do real damage would then have depended on the depth of the water. An explosive planted on the bottom could be effective well beyond the immediate blast zone. Up to a point, the water surrounding the explosive would help: water cannot be compressed, and the explosion would have forced the liquid around the center outwards in an expanding 'bubble' of water This could have crushed in the hull of the *Merrimack*, which was not armored" [1].

Gameplay: The Alligator will be in oar-rowed propelling mode and diver attack mode. The Alligator will have only one diver and one explosive to make the attack with. The students will have to destroy the Merrimack to succeed.

3.3.2 Hampton Roads, VA

Description: On the approach to Norfolk harbor, the Alligator would have had to navigate the heavily guarded Chesapeake Bay and Hampton Roads area. This location was fortified with many ships, mines, and land batteries.

Gameplay: Navigate through the bay and river to reach the Norfolk harbor area. Stealth will be essential to successfully avoiding the enemy.

3.4 Post-Project Missions

The following missions are not implemented in the game, and are provided for future development purposes. We have included two scenarios for training missions, which should include pop-up interface instructions, and a number of historical and networked scenarios.

3.4.1 Training - Diver attack

Description: Taking out a stationary unmanned frigate in a harbor.

Gameplay: The players have to destroy a stationary ship with a diver attack.

3.4.2 Training - Man-hole attack

Description: Taking out a stationary unmanned frigate in a harbor.

Gameplay: The Alligator would be in man-hole mode and the students have to learn how to use the man-hole mode to blow up the unmanned frigate.

3.4.3 Appomattox River Raid

Description: "*Alligator's* assignment to the James River encompassed both the planned attack on the bridge over the Appomattox as well as the destruction of obstacles below Drewry's Bluff. However inappropriate the submarine might have been on the attack against the railroad bridge, the underwater obstructions at Drewry's Bluff were an ideal target more suited to the abilities of 'salvage boat.' McClellan's retreat from Richmond in the course of the Seven Days' Battles meant that the bluff was again beyond the reach of the Navy, which preferred not to be trapped along the narrow river without the support of the Union army" [1].

Gameplay: The Alligator will be in oar-rowed propelling mode and diver attack mode. The Alligator will have only one diver and one explosive to make the attack with. The students will have to destroy the bridge in these modes.

3.4.4 Charleston Harbor

Description: "As per information related in Mark Ragan's 'Civil War Submarines of the North and South,' references to submarine operations in the harbor are plentiful. At one point, Admiral Dahlgren requests '3-4 submarines' to clear obstructions in Charleston harbor. In removing obstructions that were anchored to the bottom--such as tethered mines--*Alligator* could have been very effective deploying a diver. This had been done quite reliably from Villeroi's boats for several years. But how could it have had any effect upon a floating log boom or chain suspended a few feet beneath the surface? Only an explosive tied to the links or line of logs would have had an effect. The fact that Eakins was willing to attempt the mission and sounds supremely confident in his ability to execute it using *Alligator* suggests that he understood better than we do how the submarine could be used. As per the longer article, this is entirely conjecture, but Sam might well have meant to make his attack via the upper hatch of the boat. If so, the *Alligator* could have been very effective" [1].

Gameplay: The Alligator will be in screw propelling mode and man-hole attack mode. The Alligator may have up to two divers and one explosive per diver to make the attack with. The students will have to destroy the chains in these modes.

3.4.5 Drewry's Bluff

Description: "*Alligator's* assignment to the James River encompassed both the planned attack on the bridge over the Appomattox as well as the destruction of obstacles below Drewry's Bluff. However inappropriate the submarine might have been on the attack against the railroad bridge, the underwater obstructions at Drewry's Bluff were an ideal target more suited to the abilities of 'salvage boat.' McClellan's retreat from Richmond in the course of the Seven Days' Battles meant that the bluff was again beyond the reach of the Navy, which preferred not to be trapped along the narrow river without the support of the Union army" [1].

Gameplay: The students have to blow up some obstructions on the riverbed in order to learn how to use the diver attack mode of the Alligator.

3.4.6 Raid on Gosport Navy Yard

Description: "Even before *Merrimack* was specifically targeted, Villeroi offered to make a general raid on Gosport Navy Yard in Norfolk. If *Alligator* could have succeeded against the rebel ironclad, she would have stood a good chance of sinking additional Confederate vessels in the navy yard" [1].

3.4.7 Break the British Blockade

Description: "Most people do not realize how close we came to fighting World War One in 1862-1863. Even before the Trent Affair moved us close to open warfare with England, the support of Britain and France for the Southern Confederacy placed them in opposition to Lincoln's Union. Smarting from her recent loss to the French and British in the Crimean War, Russia let it be understood that it would ally itself with the North in any open conflict; in 1862, the Czar sent his Baltic and Pacific fleets on a 'good will tour' of the western and eastern coasts of the United States, with secret orders to be ready to pounce on convoys bringing troops and supplies from Europe. The Navy's capture of

New Orleans in April 1862 stayed the Europeans' hands and Gettysburg a year later ended once and for all the threat of a global war" [1].

Description: "Would a squadron of a half-dozen *Alligators*, such as proposed by William Hirst in December 1861, have been an effective weapon against an Anglo-French blockade? The same problem posed by the depth of the water in an attack against *Merrimack* would have been faced off-shore: how to attack a floating vessel from the sea floor. Did Hirst already have an inkling of an alternative way to use *Alligator*, such as Sam Eakins seemed to be hinting at in his plan for the attack in Charleston harbor" [1]?

3.4.8 Confederate attack on Union Blockade

Gameplay: As the Confederacy, capture an Alligator submarine, and use it to destroy the Union Blockade.

3.4.9 Destroy an attacking fleet

Gameplay: Approach and submerge beneath an incoming fleet with a group of Alligators. Destroy as many ships as possible before they can attack.

4 Physics

Physics in any game is quite important, governing the way bodies move and interact in the world. This section is an explanation of what steps we went through in the design of our physics system, and in the implementation of that system.

4.1 Physics Design

The design of the physics system was a long iterative process. There were three major iterations in the development of the physics system. The first was to use C4's basic physics system. The second, after determining that C4's basic physics was unable to fully accommodate our needs, was to alter the C4's physics system enough to suit our needs. Finally, we settled on using a third-party physics system called Bullet [5].

Originally, we had a design that solely used C4's collision detection and response system. We decided to go with this system because it was already in place, and we believed that we didn't need to modify it very much. Collision detection and response was already implemented in the demo code to control the game characters.

However, we ran into a major problem. The code for character movement was based upon a vertical capsule collision volume (for a bipedal character), but the Alligator is a horizontal vehicle and so needs a horizontal capsule. We thought that we could simply rotate the capsule from vertical orientation to horizontal orientation but this didn't work as we had hoped. All of the equations that are used for collision detection, sliding, and ground penetration in C4 are based on staying vertical (global +z), and so they did not work when we rotated the capsule. We realized that for us to be able to use C4's physics we would have to learn all of the code in place for the vertical capsule, and rewrite our own equations to work for a horizontal capsule. We went ahead and attempted to do such a thing until we realized that even after a complete rewrite of the equations it still did not work the way we expected. Even the creator of C4, along with some of its long-time forum community veterans, told us that it wasn't an easy feat to do the horizontal capsule rewrite [4]. Therefore, we tried other ways of producing a horizontal capsule.

We moved onto another possible solution using C4's physics. We decided that it might be possible to use a bounding box (orientation does not matter) for the Alligator submarine. After searching through C4's engine code we realized that this too wouldn't be practical. All of the collision detection and response is coded for spherical collision volumes (a capsule is just a swept sphere). Therefore, we would have to rewrite many equations for collision detection and response at a level as low as the C4 engine code. This made it very impractical for the time span of the project.

Finally, we did some research to find any other options for our physics system that might be available. We found a few third-party options. The options we found from the C4 commuity were Ageia's PhysX, Newton Game Dynamics, and Bullet Physics [4]. Although we preferred PhysX, due to its constantly updated and improved integration with C4, as well as its additional features and stability, we had to decide against it. The main reason we refused it was because it is not Macintosh-compatible (only Windows, Linux, and Playstation 3), and we were designing Alligator to be cross-platform since many schools use Macintosh computers. We ended up choosing Bullet Physics because it is multi-platform, and it already had a C4 integration (although, from a distant previous build a year earlier). Although it would take some work to reintegrate the Bullet-C4 integration code with the newest C4 and Bullet builds, we decided on using Bullet for our physics [5].

The main design of our physics system was to create a base controller class of which anything we desired to be physically responsive could derive from. We decided that there could be two main types of physics bodies, a controlled physics body and an uncontrolled physics body. A controlled physics body is any rigid body that was controlled by a C4 controller. An uncontrolled rigid body is one that is not controlled by a C4 controller, these can either be static objects or simply objects that are not static but do not have a controller (terrain, for example).

We also created a physics manager to handle the initialization of the physics system, stepping the simulation, addition and removal of physics bodies, and collision maintenance.

4.2 Physics Implementation

The foundation of the entire physics system rests in assigning physics properties or controllers to nodes in C4's World Editor. A node can be assigned a physics rigid body property or a rigid body controller. The difference is that the property is used for the uncontrolled nodes that a user wants to be a part of the physics system. The rigid body controller assigns a physical controller to the node. Note that if a property and a controller are assigned to the same body, the controller takes precedence. Then, after the property or controller is assigned, there are many settings to be set that will govern the physical characteristics of the object.

Using Bullet as our physics foundation allowed us to sidestep the problems we were having with the vertical capsule (detailed in the previous section), because Bullet allows more than just spherical collision volumes to be used. It allows many types of collision volumes, such as bounding boxes, spheres, cylinders, convex hulls, and height fields. We have exposed different collision volumes to the C4 World Editor, so that level designers may choose the collision volume based on their geometry needs.

The physics manager has three main functions it uses to assign rigid bodies to nodes: AddPhysicsToNode(), AddPhysicsToEntity(), and CreateTerritory(). AddPhysicsToNode() adds physics to a node of type geometry, while AddPhysicsToEntity() assigns physics to a node of type entity. However, the entity node does not have geometry, so this function goes through only the *first* level (breadth) of the entity node's subnodes, and applies physics to all geometry nodes in that level. The reason for this is that we wanted a way to have some parts of a model to have physics applied (the frigate model's hull) and some to only be visual (the masts on the frigate model). Also, it gives us greater control, because collision volumes are based on the geometry being applied physics, so something such as a frigate with its masts included

would give a bounding box that does not represent only the hull of the ship (which is needed for buoyancy and drag forces). Lastly, the function CreateTerritory() creates a rigid body without the need for a geometry node. CreateTerritory() creates a rigid body with a certain type of collision volume and assigns that body to a controller passed in as a parameter. Using CreateTerritory(), physics rigid bodies can be assigned to non-geometry nodes. However, territories do not behave as a physically responsive object. They are used only for collision detection and then signify an observer as to when it collided with something.

Physics are initially added to the nodes of the world in through the physics manager. First, the game world file is loaded by C4. Then, with the scene graph initialized, the physics manager traverses the scene graph, finding all the geometry or entity nodes in the level, and determining if they are assigned a physics rigid body property or a controller. If so, it then assigns physics to that node and adds it to the physics world simulation.

There is one type of physics controller that is handled differently than the rest. When the physics controller being assigned is of the type *kControllerDiver* (the diver), all rotations for that controller are constrained. The diver is a GameCharacterController that must be able to walk along the terrain, controlled by the player. Our first implementation of this controller resulted in some problems, however. At first, we constrained the diver's rotation to only be able to be rotated along the z axis (locking rotation in the x and y axes). We restrained this axis so that the diver always remains upright and doesn't fall over when moving. A problem came about that because rotation along the z axis was still allowed, when the diver walked alongside a sloped surface the friction between the objects rotated the diver in an unexpected way. The diver was always rotated, automatically, to be orientated uphill of a sloped surface. This caused control of the diver to be unreliable and even frustrating. Therefore, we then locked all rotation of the physics body. We would rotate the C4 model, and not the physics body, to always point in the direction that the game camera is facing. Finally, we moved the diver by setting its linear velocity rather than applying forces or impulses (applying velocities is independent of mass and the effects of gravity, granting us greater control of movement). By

combining all these changes to the physical representation of the diver, we obtained the control of the diver that we were searching for.

When rigid bodies are created in the physics world, they must be managed. The physics manager keeps track of all the rigid bodies that are created and added to the world, as well as whether or not the rigid body is controlled or uncontrolled. Keeping track of bodies in this manner allows us to remove a specific rigid body from the world if we so choose, such as when it is destroyed. Also, management allows us to decorate, or wrap our physics rigid bodies with some other properties. One very important property is whether or not the rigid body takes place in collision response or is simply involved in collision detection. When a controlled node is involved in a collision its HandleCollision() function is called with the node it collided into. The HandleCollision() function is also called even for nodes which are not involved in collision response, as long as they are controlled. This control, whether or not to apply physical responses, is obtained by using the customNearCallback() function. The callback function is invoked when Bullet determines that a collision occurs, and reports between what two bodies the collision occurred.

There is another feature that we have implemented in the game. We have created the ability to view the bounding volumes for each of the physics bodies. Every time a rigid body is created, a viewable collision volume is created to match the rigid body, using a wireframe outline. Thus, a user can enter the console command “bounds” to toggle viewing the collision volumes for all of the nodes in the world.

A derivative of the PhysicsBodyController class (the base class for all the physics controllers) is the GameCharacterController class. This class offers some functionality to the rest of the game controllers, providing variables and functions to control the physics body. It has variables for thrusting, stopping, strafing, and some righting forces (some of which are not used). The pitch and roll righting forces, try to orient the body towards the up vector (+z), either in the pitch axis or the roll axis. The GameCharacterController

class also has the functionality to apply water forces, such as water currents, buoyant forces, and drag forces.

The water current forces are applied by using the water current chart stored in the GameData class. The current forces are stored as blue and green color values, which are translated as vectors. The vectors are translated with green representing the vertical component and the blue representing the horizontal component. Note that color values range from 0 to 255; to obtain both negative and positive values for each color we had to move the zero-point up to the color value 128. Thus, numbers from 0 - 128 are negative, with 0 being the highest negative current force one can have. Numbers from 128 - 255 are positive, with 255 being the highest positive value one can have. Then, the 0 - 255 color values are scaled from 0 to 1. For instance, the color value (0, 255, 255) will get translated to (0, 1 1). The color value (0, 64, 192) will get translated as (0, -0.5, 0.5). Also note that the vectors obtained from the color values in the current charts are not only directional vectors, they have magnitude also. The greater the length of the vector is the greater the force that the current has. However, the vector is not a force vector in that it does not have the units of a Newton (unit of force). Therefore, the current vectors have to be scaled by a force magnitude which has the Newton units. This force magnitude is stored in the configuration file for each mission under the variable name "maxCurrentForce". Setting this value will scale the physical forces that the currents get applied. Note that color values that are either 0 or 255 will get the full current force applied and so will apply the greatest force on the objects it affects.

The buoyancy force works in a completely different way. Buoyancy forces are applied according to Archimedes' Principle, with the equation [6]:

water_density * volume * gravity_acceleration

This equation applies a force opposite to the gravity acceleration, according to how much water is displaced by the volume. Every rigid body has a collision volume, and every collision volume has a size, stored as a 3-dimensional vector. This size vector can be

used to determine the volume of body. To apply the buoyancy force, we must know how much of the body's volume is submerged in the water. We used approximations to determine this number. We assume that the volume is not rotated in any way and sits in the water perfectly level (an accurate approximation because bodies have righting forces to keep them aligned in such a way). This assumption allows us to determine what percentage of the body is submerged in the water, by determining how deep the lowest point in the body is below the water level. The volume of a ship is a bit different, however, because a ship uses a bounding box volume, but the hull of a ship does not completely fill up the bounding box. Rather, the hull of a ship is more triangular, and so what we do to adjust for this is to determine the depth of the bottom of the ship at rest (the draught), and then use that value to approximate a triangular volume.

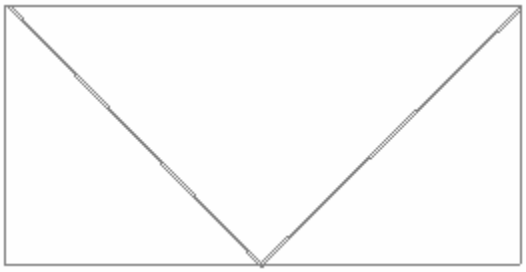

Figure 8 – Ship Bounding Box

In Figure 8, the outer box represents the bounding box of the ship. The inner triangle is the triangle created based on the depth of the hull.

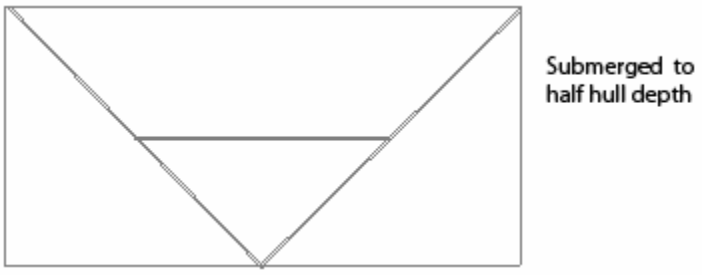

Figure 9 – Half Submerged Ship Bounding Box

As shown in Figure 9, the triangle created by submerging the ship's hull to half its total depth gives a triangle that is much less than as if we had used just half the bounding box volume. This gives a better approximation of how much of the ship's hull's volume is submerged.

The drag force is similar to the buoyancy force. It is calculated according to a modified Lord Raleigh's drag equation [6]:

0.5 * water_density * dragCoeff * mass * velocity * velocity

The main thing to know about this equation is that it is dependent on velocity squared, so the faster one is traveling, the more drag that is applied. However, due to our settings for the water density and drag coefficients, the equation is simplified to:

0.5 * mass * velocity * velocity

We determined these values by experimentation, and to keep the necessary values of mass lower than truly needed. If true water density were used, the masses of objects would have to be 1000 times higher than they currently are due to the volumes of our objects. Extremely large and extremely small values are avoided in physics simulations, due to precision errors and type memory limits. It is therefore common to normalize physics properties such as mass and forces into manageable ranges.

For more detail on the physics system, please review the Technical Manual, which is found with the project files.

5 Artificial Intelligence

Alligator utilizes the steering behaviors provided by the OpenSteer library. Created by Craig Reynolds while at the Massachusetts Institute of Technology, OpenSteer is a C++ library created to help construct steering behaviors for autonomous characters in games and animation [7]. All of the moving objects in the OpenSteer scene are represented by the Vehicle class. OpenSteer calculates the velocity and direction of movement for these vehicles, and these values are used to determine how these objects are being moved in the game. Currently, only the AIShipController is connected with OpenSteer. The AIShipController has a sensor, which determines whether the ship can see the player. If the player is spotted, it makes decisions to maneuver or fire, based upon a state machine design.

5.1 Artificial Intelligence Design

We made the decision of using OpenSteer during the early stages of the development process. We believed it would provide us with many features required by the project. OpenSteer enables the AI objects to steer to a specific location, chase an object, evade an object, avoiding collision between other AI objects, wander, and provides turning radius and speed interpolations [7].

There are a few examples of combining simple steering behaviors to produce more complex behavior in the OpenSteer library. At first, we examined the "soccer" and "capture the flag" demos. Soccer was a simple example, which we were able to integrate with the C4 engine easily. Capture the flag was a more in-depth example; it demonstrates a group of enemies trying to stop an entity from capturing the flag. This demonstrated many of the behaviors we desired in the game AI. However, the drawback of this example is that the collision avoidance implementation for non-spherical objects is minimal, similar to our problems with the C4 collision system.

Collision avoidance is an important part of designing any AI system. We initially decided to use a flowfield system, in which a pre-generated vector map is used to

influence the movement of the entity [8]. The flowfield can be constructed to steer entities away from level geometry, and is also used for the water current forces. This solution alone produced suboptimal results, with the entities making physically impossible movements at times. Over time and through research, we learned enough about OpenSteer and steering algorithms in general to utilize the MapDrive system in OpenSteer, allowing a vehicle to steer away from obstacles with irregular shapes, speed up when in open areas, and slow down when approaching obstacles [9].

Sensing is a process which has undergone several iterations in our design. In the game, sensing determines how the enemy ships may spot the Alligator. At first, we performed a simple distance comparison between the AI ship and the submarine, and this was affected by some environmental factors such as fog and rain. This system was too simplistic, and we decided to use an invisible rotating physics collision volume that would scale in size according to the weather conditions.

The decision making system for the AI is a basic state machine, which decides whether the ships should wander, chase, or attack. The conditions are based upon whether or not the Alligator is spotted, and the submarine's position relative to the ships location.

5.2 Artificial Intelligence Implementation

The implementation of the artificial intelligence consists of several areas: navigation, collision avoidance, agents, and sensors. Navigation refers to steering the AI object correctly. Collision avoidance means avoiding the other AI objects, non-AI objects, and terrain. Agents refer to the AI object decision making mechanism. Finally, sensors dictate how the AI object "sees" other objects in the world, namely the Alligator submarine.

The first step in implementing the AI system was the integration of C4 and OpenSteer. Three scene graphs are maintained, one each for C4, OpenSteer, and Bullet physics. OpenSteer determines how an object should be moved, and returns kinematic values which include the velocity and orientation of movement, common to most steering and

movement algorithms [9]. These values are passed to the physics engine, and update the physics scene graph, which in turn updates the C4 scene graph. After these values are passed into the physics engine, we ensure that the OpenSteer scene graph is synchronized by resetting the position in the OpenSteer scene with the most recent physics object position.

When a developer places an AI entity in the game world, an OpenSteer entity is generated automatically, and its OpenSteer updates are performed in the Move() function of the entity's controller. In the future, if the developers would like to create a class which can use the functionality from OpenSteer, they may create a new controller class which represents the C4 object, and subclass the existing AIShipController. Alternatively, the base VehicleController class may be used to create an AIVehicleController, and this may be inherited from to provide artificial intelligence behavior. Please refer to the EnemyShip class and the AIShipController class for more information.

The OpenSteerManager class is a singleton which consists of a list of OpenSteer objects, and also provides functions to convert between the C4 coordinate system and the OpenSteer coordinate system. The TerrainMap class represents the terrain of the world. All of the OpenSteer objects need to read from the map in order to avoid running into the land.

To navigate to a certain location, we only need to pass in a point, and OpenSteer automatically calculates the velocity and orientation. Note that the coordinate space system is different between C4 and OpenSteer; the z axis in C4 is up, while in OpenSteer the y axis is up. Since the ships only navigate on the water surface, the height value is ignored in the OpenSteer scene graph most of the time.

Collision avoidance is adapted from the MapDrive demo included with OpenSteer. In order for the ships to know where obstacles are, we need to generate a two-dimensional image which consists of two RGB colors. When RGB color values are 128, it means that

area is open for navigation. While RGB are greater than 128, it indicates that the area is blocked.

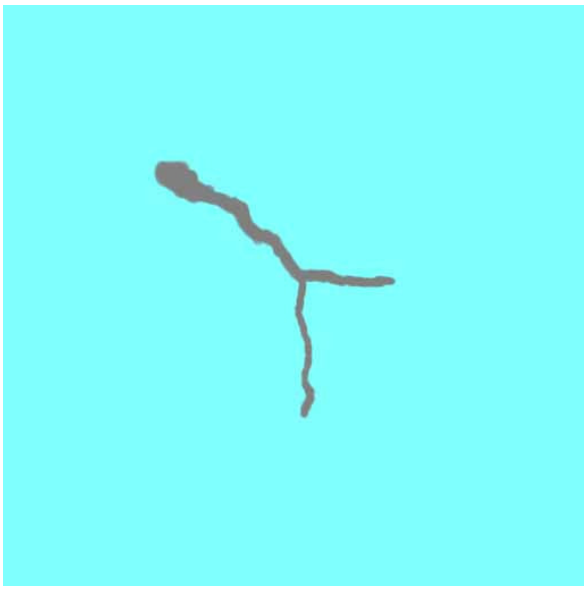

Figure 10 – Collision Avoidance Map Image

This image, as shown in Figure 10, is converted into a list of boolean values which can be quickly accessed by the MapDrive implementation. The MapDrive system projects several rays from the front of moving objects, used to scan the map so that the system can find the closet blocked region, and perform the algorithms necessary to avoid collisions. Therefore, it is critical to have a correct and precise bitmap image representation of the world. For more details of the MapDrive system, please refer directly to the OpenSteer library's MapDrive plug-in [7].

The actions that the ships take depend on the agent, a state machine consisting of a wander state, chase state, and attack state. All of the ships start in the wander state, and wander within a given distance from their starting locations. When the sensor of the ships detects the Alligator, the state is changed to the chase state, and they chase the submarine to reach attacking range. When they are close enough, they change to the attack state, and commence aiming and firing while maneuvering around the target.

When they lose sight of the target, they return to the wander state around the location where the Alligator was last seen.

For more details on the artificial intelligence system, please review the Technical Manual, which is found with the project files.

6 Art Assets

Since the project aims to offer a historically accurate representation of Civil War submarine operations, a realistic art style has been used throughout the game. We felt that this choice enhances the experience of the historical conditions that occurred during the Civil War, in particular to naval and submarine combat. For instance, poor visibility, cramped quarters, and low light levels were particular issues that we addressed in the artistic design.

6.1 Art Design

Art creates a major part of the appeal for any game title. Even though our game is not aimed at the mass market but at middle and high school students, there are many other aspects of art in games that makes it a very integral part of the design. What the players see and hear in the game, and what their focus is drawn to, is largely dependent on how the art design is handled.

From the earliest stages of the project, our sponsor stressed the importance of historical preciseness in the game, and how it was necessary for the players to feel the hardships encountered by the captain and the crew of the Alligator. We took this requirement very seriously, and set out to meet it with a realistic and historically accurate art style. We used several sources to gather blueprints, images, drawings, descriptions and various other details on the important art assets that needed to be built [1, 3, 10]. For the most important object in our game, the Alligator, our sponsor offered us a great number of resources, including a 3D model he made himself. He also provided images of original maps and drawings for the locations where the Alligator's missions were to take place, other drawings and blueprints for the various kinds of vehicles that would appear in these missions, and various images of Civil War era housing. To complement this data, we performed our own research on the design and look of objects such as the cannons, guns, ships, and clothing. We believe that extensive research was necessary to create in our minds the foundation of the right look and feel we would utilize for this game.

To achieve the level of realism we set out for, not only did we need to know what our objects would look like, but also what their sizes were, what kind of materials were used to build them, and how they operated. Our sponsor's own website proved to be an invaluable source of information on the Alligator and Civil War Navy clothing [1]. We were able to find the dimensions of many of the key vehicles (such as the CSS Virginia) on Wikipedia [10]. Once we found what the materials the vehicles were built from, and what the overall color of these materials were (note that most reference images were grayscale, which yields very little information about the color of an object), we looked for photos on texture sites such as CGTextures [11] and Mayang's Textures [12] that would help us simulate the correct look of these objects. Given the limited amount of time we had for this project, we concluded that it would not be feasible to take our own photos for texture work as that would have taken too much time and effort.

Admittedly, the first iteration of a model or texture rarely achieved the goals of the project. The weekly meetings held were excellent opportunities for our sponsor and advisor to provide input on how the textures would need to be adjusted and where the models were incorrect; such input turned out to be invaluable in maintaining adequate system performance and historically accurate assets. Some assets, however, could not be perfected the way they were meant to be due to time limitations as stated above. Another benefit of hearing our sponsor's and advisor's opinions on the art assets was that they provided us with different perspectives than our own, which aided us in challenging and judging the decisions we made regarding these assets.

6.2 Art Implementation

We knew from the project's conception that we would not have the time to make our art assets perfectly polished, due to the limited amount of time that we had for this project. However, we still needed to stay truthful to the historic descriptions of the entities that we were to model. Therefore, we had to make trade-offs to properly distribute our time between working on primary art assets, their details and textures, secondary art assets, and other assets that were needed to fill the game world.

Sub Exterior Model	Fish Animations	**Chain Net Model**

Sub Interior Model	**Shark Model**	**Rifle Model**
Sub Animations	**Shark Animations**	*Explosive Model*
Frigate Model	**Seaweed Model**	Skyboxes
Frigate Animations	Tree Model	Interface Icons
Pier Model	*Picket Ship Model*	Interface Windows
Dock Model	**Sloop Animations**	Interface Fonts
House Model	Smithy Model	**Title Level**
Drydock Model	Shipwreck Model	**Training Level**
Human Model	Lighthouse Model	Mission Level 1
Human Animations	**Farm Model**	*Mission Level 2*
Diver Model	Cannon Model	**Mission Level 3**
Diver Animations	**Fort Model**	**Mission Level 4**
Mine Model	*Gun Battery Model*	**Mission Level 5**
Fish Model	Cannonball Model	

Table 1 - Art Assets Listing

The initial art assets list was created as shown in Table 1. All of the assets necessary for the prototype have been completed. The assets that we did not have time to complete are shown in bold in the table above. The assets that can be considered partially complete and ones that have placeholders in the game are shown in italics.

After examining the number of assets that were completed, it is obvious that we initially overestimated the amount of work that we could possibly complete in the duration of this project. This occurred for several reasons including the C4 engine workflow and physics integration.

First of all, even though we were quite familiar with the basic art assets workflow for the C4 engine, we tried using many different features in this project that we have not used before. Most of these required learning various new ways to export the assets that we created in external applications, and testing this new workflow for consistency and accuracy. An example of this was the terrain workflow which is discussed later. Some

of these processes had very little or no documentation and figuring out the various quirks proved to be extremely time consuming. We sought help from the C4 engine experts on the C4 engine forums in cases where we encountered difficulties in both asset creation and code implementation [2].

Midway through the development of the game, we started using the Bullet physics engine, and the integration did not work immediately with the assets we had already made. It took many hours of testing and debugging to understand and finally resolve the issues regarding the connection between the models and their physics properties. One example of this is how a model's pivot point affected its bounding box and other physical properties in the game, such as buoyancy levels.

We did not have a complete comprehension of the amount of effort involved in making an art asset entirely game-ready. Since we only had one person working on modeling and texturing, and another on importing and animation, a great amount of time was spent on the primary assets before they could be placed in the game world and work perfectly.

The primary application we used to create these assets was Maxon's Cinema4D. It is a 3D modeling and animation suite that is known for its ease of use and versatile and logical workflow. Integrated into it is Maxon's BodyPaint3D, which is the industry standard software for UVW editing and texturing; we used this application along with Adobe's Photoshop CS2 to create all the texture maps for our 3D models. One issue with using Cinema4D is that it currently has no COLLADA import/export support, which made it difficult for us bring the assets created in Cinema4D into the C4 Engine. We overcame this issue by using Autodesk's Maya and 3DSMax applications, which have COLLADA import/export plug-ins, as middleware plug-ins. We also used Maya for animating the art assets, as the animations created in Cinema4D were not preserved properly across the different file formats, and Cinema4D does not currently have the Full Body Inverse Kinematics (FBIK) system that Maya offers. Pixologic's Zbrush3 was used to finalize and add detail to level terrain. Even though we would have also liked to use Zbrush to create normal maps for some of our art assets by projecting high-poly

meshes on to their low-poly counterparts, that would have taken too much time and experimentation that we could not afford.

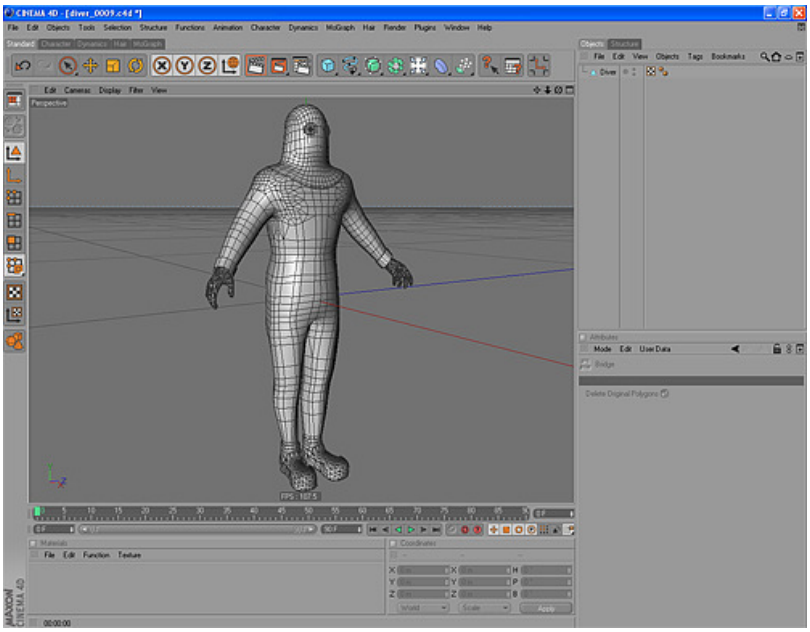

Figure 11 – Diver 3D Model Created In Cinema4D

The workflow particular to the C4 engine is slightly cumbersome, and so we will explain the required steps for importing a fully textured and animated model. The initial step in any modeling is to gather reference material (images and descriptions) for the desired object. Then the object is modeled in Cinema4D using the reference material, keeping in mind the polygon-count in relation to the object's importance in the game, how many instances of the object can possibly appear on the screen at once, how the polygon flow will affect the UV mapping and texturing, and where normal map detail will be used in place of actual geometry. The finished geometry model for the diver is shown in Figure 11.

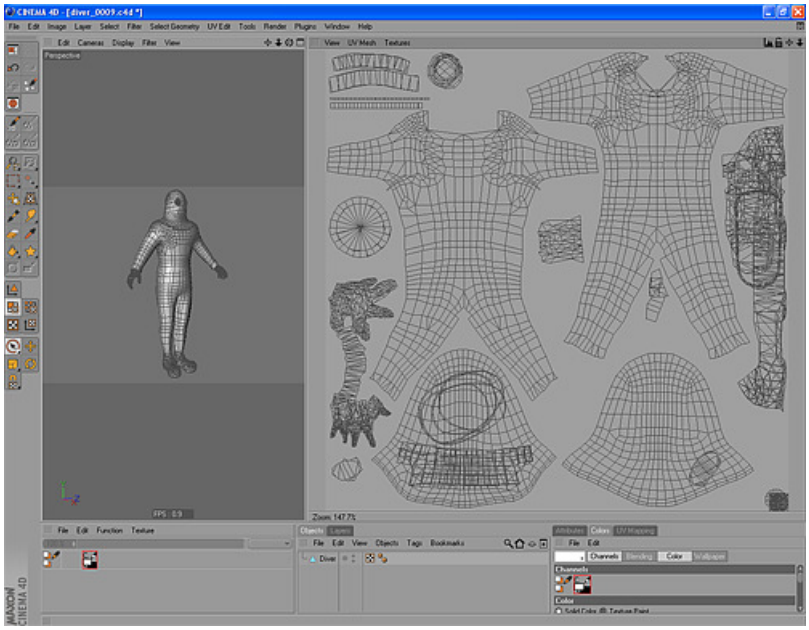

Figure 12 – UV Mapping of the Diver Model

Once the modeling is complete, the object is UV-mapped in BodyPaint3D, as shown in Figure 12. Factors are taken into account such as the possible use of UV overlapping to save from texture space, ease of seam removal by using Projection Painting [13].

For texturing, a snapshot of the UV is taken from BodyPaint3D in double the size of the final texture into Photoshop and an initial search is done on texture websites for source images that can be used to create the right texture. Afterwards, the source images are manipulated in size, color, orientation, and detail as necessary, and used along with others to create an initial texture, as shown in Figure 13. This first iteration is applied onto the model in Cinema4D, and is inspected to find areas that need fixing.

Figure 13 - Diver Texture

The seams are corrected using Projection Painting with the Clone brush, and the texture is finalized in Photoshop. Since the texture was created in double the size it is going to be used in the game, it is resized to half the size and then sharpened using the Smart Sharpen filter in Photoshop (usually with the Radius set to 0,5-1,5 pixels and Amount set to 60-120%). If necessary, a height map (bump map) is created in Photoshop, in most cases by modifying the original texture or using other source images. Both the color map and the height map are saved in Targa (TGA) format, which is the format that C4 uses to import textures into its own proprietary TEX format.

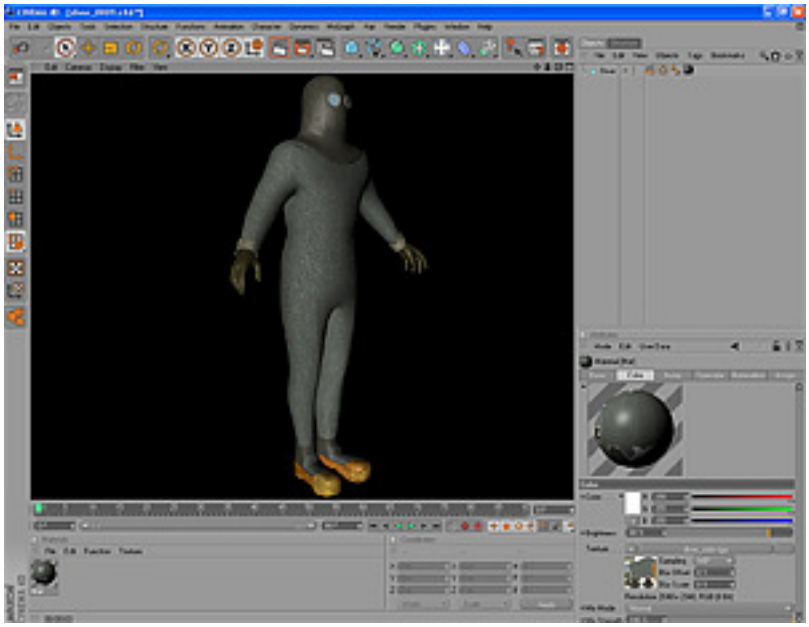

Figure 14 – Texture Applied to the Diver Model

When the model and textures are finished, as shown in Figure 14, the model is exported from Cinema4D in Wavefront (OBJ) format. This OBJ file is then imported into Maya and the model is rigged using joints and FBIK. Bone weights are fixed if necessary in Maya, which control which areas of the mesh model are moved by each joint. The rigged and animated model is exported from Maya in COLLADA (DAE) format, which is the format the C4 Engine uses to import all geometry. The textures are then imported into the engine, and a new world file is created where the imported texture is used to create a material, and that material is applied onto the imported model. We saved these world files as reference worlds that can be placed with markers for static scenery in the game. Finally, this world file is exported to a C4 model file (MDL) if it is used for an animated entity, as shown in Figure 15.

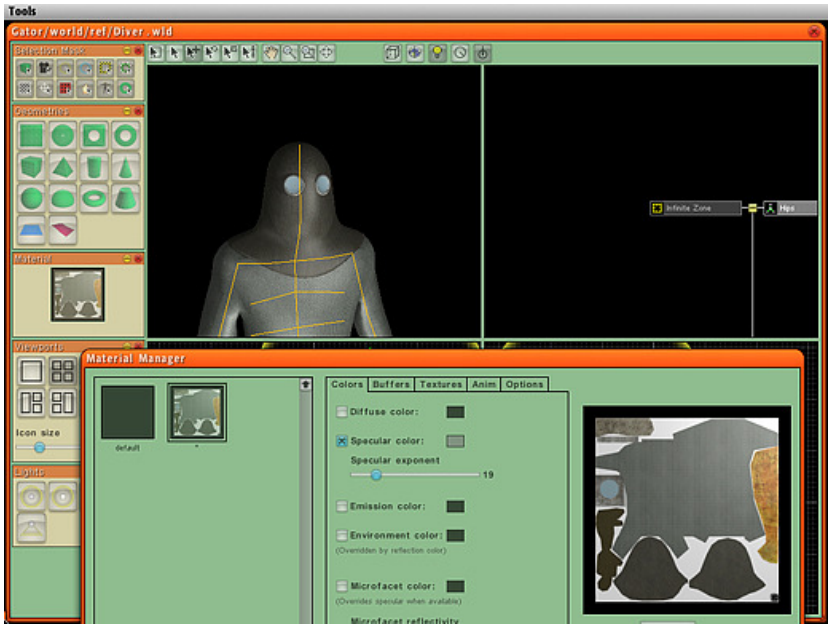

Figure 15 - Diver Model Imported Into C4 Game Engine

This workflow is more or less the same for almost all of the assets we created, except for the terrain. The terrain was one of the elements of the art implementation that proved extremely difficult to create correctly. This is mainly due to the currently poor support for terrain in the C4 engine (which is going to be addressed in the future Fireball releases). There is an Import Terrain tool in the World Editor of the C4 Engine that creates a 3D terrain from a height map. This tool takes a long time to generate the terrain when supplied with height map textures of size 1024 by 1024 pixels or greater, upwards of ten to fifteen minutes if it succeeds at all, and the terrain it generates does not have the quality one would expect. It also does not leave flexibility to create terrain level-of-detail models or decrease the polygon count of the terrain in any other way. A member of the C4 Engine forums posted a code snippet that helped create a smoother and somewhat higher quality terrain, yet this still was not up to our expectations [2]. Therefore, we decided to try different methods using various applications. We tried using Freeworld 3D by Soconne Inc. [14] and Terragen [15], but decided in the end to use Cinema4D and Zbrush.

We started out with the Digital Elevation Model (DEM) maps that our sponsor provided us with. We used the Displace Deformer on a 16,000 polygon plane in Cinema4D to

create a 3D terrain, much like the C4 Engine generates the terrain from a height map. This gave us a rough model to work with, as shown in Figure 16.

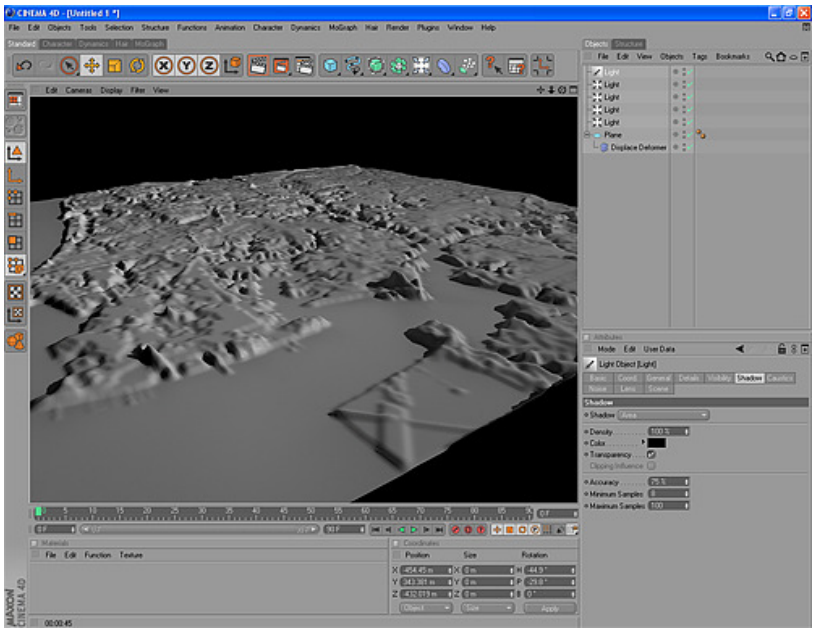

Figure 16 - Terrain Generated in Cinema4D

The flat areas in the model are actually the level of the sea, instead of the sea bed, so we needed a way to carve the flat areas downwards to create the seabed ourselves. Thus, we exported this model to the Wavefront format to be brought into Zbrush. Once the model was in Zbrush, we carved the waterbed, added some detail to make the terrain look more interesting, and flattened certain areas to allow for easier population by houses and trees. After that, the model was exported out of Zbrush, again in Wavefront format and brought back into Cinema4D for texturing and scaling.

The initial texturing for the terrain was done in Cinema4D using a procedural material that allowed us to use a seaweed texture for heights from seabed up to shore-level and a grass texture for the rest. Then we took this texture into Photoshop and added stone textures for the slopes of the hills and other types of variation to make the terrain texture more interesting. After that, the terrain texture is applied on the model, the model is subdivided into 16 equal parts to make collision calculation easier for the engine, and the

terrain model goes through the same pipeline as any other art asset to be finally brought into world in the C4 engine.

Audio content creation for the game proved to be less problematic than geometry. We used Cubase SX 2.0 to sequence, record, and adjust levels and effects. The sound effects for the game were taken from various free websites of sound clips, and adjusted in Cubase to reduce noise, apply effects such as reverb and delay, and output 16-bit WAV files. The music for the game was also written by the team, based upon MIDI files found by the sponsor of Civil War era songs. Since MIDI files only contain note and controller information, they are used to drive software and hardware based synthesizers, samplers, and other programs. We arranged the music for orchestral instruments, and recorded these arrangements with Native Instruments Kontact 2 orchestral sample library, producing professional and realistic music for the game.

To use sound in the C4 engine, we had to convert the raw WAV files generated by Cubase into compressed WAV files with Audacity, a free sound editing program. C4 uses compressed WAV for larger files, and uncompressed WAV for small sound files. Both types may be streamed in real-time, instead of pre-loaded during level loading.

7 Subsystem Design

7.1 Game Subsystem

The Game subsystem consists of the core classes for basic game functionality. The world, cameras, multi-purpose controllers, input, and game data are maintained by this subsystem. The Game class is the main application class, which handles the rendering loop, updates for physics and AI, the current camera, and loading worlds.

7.2 Bullet Physics Subsystem

Physics simulation for the game is located in the Bullet Physics subsystem. These classes provide all of the physics implementation for the project, and most of the variables are exposed to the C4 world editor.

7.3 Interface Subsystem

The Interface subsystem provides the graphical user interface and configuration windows for Alligator. All of the windows, buttons, and text for the game are located in this subsystem.

7.4 Entities Subsystem

The Entities subsystem contains the bulk of the code for the project. The entities control all animated models in the game, including the vehicles and crew. Artificial intelligence functionality is also located in the Entities subsystem.

7.5 Weapons Subsystem

The Weapons subsystem includes all of the weapons in the game, such as cannons, mines, and explosives. The weapons are C4 entities, with functionality to deal damage and play effects.

7.6 Effects Subsystem

All particle effects in the game are located in the Effects subsystem. These include smoke, explosions, rain, and snow. Day and night cycles, clouds, stars, and lighting are also included in this subsystem.

7.7 MissionPlanning Subsystem

The MissionPlanning subsystem controls the mission configuration and level variables in the game. It provides path creation via waypoints, water current and depth grids, weather and time settings, and the interface for the Mission Planning mode.

8 Entities Functional Design

The bulk of all the code for entities occurs in their contoller's Preprocess() and Move() functions. Preprocess is called before the first frame of the game is rendered, and most initialization code is located there, or in the constructor. Move is the per-frame call for controllers, where most of the entity behavior lies.

All entities may be animated with a SetMotion() function. Entities may also implement the virtual Die() function of the base GameCharacterController, to provide custom death animations, effects, and physics changes.

8.1 GameCharacterController

The GameCharacterController serves as the base class for all character entities in the game. Inheriting from the BulletC4PhysicsBodyController, this class provides rigid body physics, hit points, a weapons list, and other base functions and variables for the entities.

8.2 VehicleController

The VehicleController is the base vehicle class for the game. All of the vehicles, such as the submarine and ships, inherit from this class. In the future, this class will allow for mounting and changing vehicles, and other functionality such as healing or the distribution of arms and ammunition.

8.3 CrewController

The CrewController contains the base crew functionality for the submarine. This includes command input and responses, fatigue settings, and carbon dioxide poisoning levels. The game will end if all of the crew die due to carbon dioxide levels.

8.4 AIShipController

The AIShipController provides OpenSteer movement and steering behaviors, cannon aiming and firing, and searching functionality. The sight of the AIShip takes into account all weather variables and the proximity of the submarine to the surface.

8.5 StationaryObjectController

The StationaryObjectController is used for the Merrimack (CSS Virginia) ironclad, the mission target in the Norfolk mission. This class is an entity without AI control, and may be subclassed for other destroyable objects such as bridges.

8.6 SubController

The SubController manages the player-controlled Alligator submarine. The submarine has ballast and air supply variables and functionality. The weapon for the submarine is currently the diver, which is created when deployed, and destroyed when retrieved. Other weapon systems such as the man-hole explosive may be added in the future. Destruction of the submarine results in game loss.

8.7 DiverController

The DiverController is deployed from the submarine, and provides a moveable diver with an explosive "torpedo", used to destroy stationary targets. The diver currently has several animations as well, for walking and death.

8.8 SubRowerController

The SubRowerController is used to control the rowers for the submarine. The rowers provide propulsion and ballast control for the Alligator. This class also provides several animations, for rowing and death.

9 Initial Timeline

This timeline is our first proposed schedule created during the design phase, before implementation and asset creation were started. This formed the basis for our detailed Microsoft Project file schedule, which is found with the project files.

9.1 Design Schedule

Table 2 lists the initial game and technical design schedule for the game.

Date	Task
Aug. 25	Start initial proposal
Sept. 8	Present proposal / Start preliminary design. Design the website.
Sept. 15	Present preliminary design / Start critical design
Sept. 22	Present critical design / Start final design
Sept. 29	Present final design

Table 2 - Design Schedule

9.2 Implementation Schedule

Table 3 details the initial technical implementation schedule for the game.

Date	Task	Pre-requisite
Sept. 10	Start coding on Game and Interface subsystems. Most important: • GatorGame • GatorCamera • GatorInterface	Use the SimpleChar Demo.
Sept. 17	Start coding on MissionPlanning subsystem Most important: • GatorMap	Game, Interface

	• GatorMission • MapPath	
Sept. 24	Start coding on Entities subsystem, Finish on Game, Interface and Planning. Most important: • GatorCharacter • Ship • Sub • NPC • PC	Game
Oct. 8	Start coding Effects and Environment subsystems, Finish on Entities.	Game, Entities
Oct. 22	Start coding on Weapons, Finish on Environment Most important: • GatorWeapons • Explosives • Cannon	Entities

Table 3 - Implementation Schedule

9.3 Assets Schedule

Table 4 contains the initial art assets schedule for the game.

Date	**Task**
Sept. 29	Submarine (Exterior) model Pier model Drydock model Mine model Skyboxes(ongoing) Interface(ongoing)

	Mission 1 Level (ongoing)
Oct. 31	Submarine (Interior) model Submarine animations Frigate model and animations Human model and animations Diver model and animations Seaweed model Tree model Sloop model and animations Shipwreck model Cannon and cannonball models Weapons models Biological models All levels (ongoing)
Nov. 30	Music Sound effects All models done All levels done

Table 4 - Assets Schedule

9.4 Testing Schedule

Table 5 lists the initial testing schedule for the game.

Date	**Task**	**Pre-requisite**
Nov. 5	Testing begins. Code freeze, no new features! Only bug fixes.	Implementation
Dec. 3	Testing finished. Must be no critical bugs!	
Dec. 4	Start final project report.	Testing finished.
Dec. 13	Report finished, last day of B term, project due.	

Table 5 - Testing Schedule

10 Project Results

10.1 Methodology

A project of this size and complexity is often handled best with iterative design and development. We revised our design of the game and the project schedule over several weeks of iterations. During development, we continued iterating on the design as unforeseen problems arose. The final class structure is greatly changed from our past iterations, due to the difficulties we experienced with collisions, physics, and AI.

We also worked together as a group as much as possible, indeed for the majority of the project. The benefits of pair programming became clear immediately for several of the complex subsystems. While each person naturally specialized in certain areas of development, we all learned to work with other parts of the project through group collaboration. This proved invaluable and lessened the time required to correct issues, since multiple people could attack the problem at once, or fix errors noticed on reading through code.

10.2 Successes

We achieved sufficient success with the design of the game systems and the overall design of the game. With future plans in mind for the title, we designed a thorough plan and vision for the game, with scenarios for missions, features, and development of the project. We have completed a viable prototype for a unique game that will be further developed into a professional product.

Physics was a stumbling block for a good portion of the project. The current physics implementation is now quite robust and extendable, and provides a solid rigid body simulation for the game. We strove to make as many of the physics settings available in the C4 world editor to ease the burden on level design.

The art assets further the realistic and historical feel of the game, through detailed textures, normal maps, and specular maps. All of the models in the game were made as historically accurate as possible. The animations, while few due to the time constraints, are detailed and convincing.

10.3 Problems

The amount of time and work needed to produce a finished three-dimensional game as complex as Alligator is beyond the scope of any single MQP. The greatest problem we encountered during the development of the project was scope; we laid out so much detail in the design phase, necessary for the final game, but all together too much work for the project length. Attempting to code and design assets for everything in the design caused the core gameplay to suffer.

Collisions were another overwhelming issue. The C4 engine is designed for vertically aligned capsule colliders (biped characters), and currently has no support for other types such as vehicles like the Alligator. We struggled for several weeks to get around this issue before we found the Bullet Physics Library. The time necessary to integrate Bullet further hampered collision and entity movement. We succeeded, but the time lost hurt the project's progress.

Artificial intelligence is often one of the most difficult aspects of game development. While we identified the OpenSteer Library early on in the design phase, integrating it has proven to be troublesome. Testing the AI movement and steering was a constant issue, and new issues arose almost every day. Due to the complex movement issues, testing aiming, searching, and firing code were also very difficult. Coupling of the OpenSteer code with our entities also proved to be a constant problem.

10.4 Analysis

Upon reviewing the project, we have identified many things we would have done differently from the start:

1. Remove OpenSteer and write our own steering and movement code. The techniques were overwhelming when first researched, but now we feel we could make a simpler system that would work more predictably.
2. Remove Bullet Physics and replace with PhysX by Ageia. Bullet was used strictly for its cross-platform capabilities, but does not perform adequately, and the integration code was out of date. PhysX is much faster, more stable, well integrated with C4, and provides many more features.
3. Design the mission terrain much earlier. We tested for most of the project with small test levels. Due to issues with terrain geometry in C4, it took us a long time to figure out how to properly create and import the mission terrain. With the actual levels created early on in the project, we could have tested, populated, and designed the missions much more productively.
4. Reduce the scope of the project. We should have focused on one or two levels from the start, and the basic gameplay and core requirements. If we had prioritized the requirements from the start, we could have avoided many of the scope issues.

11 Conclusion

By using state-of-the-art technology and historically accurate data, we created a submarine simulator prototype that can effectively represent the conditions of the Civil War Era, placing the player in the role of the captain of Alligator. The player can experience the difficult conditions under which the submarine was operated, as well as learn to coordinate with others to overcome the difficult obstacles throughout the game. This group involvement element makes the game an ideal tool for classroom education, and the historic precision allows it to become an instrument for demonstrating Civil War submarine warfare.

We encountered many unforeseen issues, forcing us to redesign, adapt, and improvise solutions. The vast scope of a completed title proved to be beyond the reach of a single Major Qualifying Project. We therefore focused upon a viable prototype with two basic playable levels for the sponsor to use for further development.

We sincerely hope that Alligator will reach completion in the near future, and help teach students and others about the Civil War and submarine warfare. We are honored to have been a part of this project from its initial conception and design.

12 References

1. Navy and Marine Living History Association. http://www.navyandmarine.org.
2. National Oceanic and Atmospheric Administration. http://www.oceanservice.noaa.gov.
3. Ward, Geoffrey C. “The Civil War”. New York, New York. Alfred A. Knopf, Inc., 1990.
4. C4 Game Engine. http://www.terathon.com/c4engine/index.php.
5. Bullet Physics Library. http://www.bulletphysics.com/Bullet.
6. Rabin, Steve. “Introduction to Game Development”. Hingham, Massachusetts. Charles River Media, 2005.
7. OpenSteer. http://opensteer.sourceforge.net.
8. Rabin, Steve. “AI Game Programming Wisdom 3”. Boston, Massachusetts. Charles River Media, 2006.
9. Millington, Ian. “Artificial Intelligence for Games”. San Francisco, California. Morgan Kaufmann, 2006.
10. CSS Virginia on Wikipedia. http://en.wikipedia.org/wiki/CSS_Virginia.
11. CGTextures. http://www.cgtextures.com.
12. Mayang’s Free Texture Library. http://mayang.com/textures.
13. Projection Painting in BodyPaint3D. http://www.maxon.net/pages/products/bodypaint3d/highlights/projection_e.html.
14. Freeworld3D. http://www.freeworld3d.org.
15. Terragen. http://www.planetside.co.uk/terragen.

Appendix A – Glossary of Terms

Bullet – The Bullet Physics Library, an open-source cross-platform game physics library. Bullet was used as the rigid body physics engine for Alligator.

C4 – The C4 game engine, developed by Terathon Software, is a full professional game engine written in C++ and OpenGL. It runs on Windows, Macintosh, and Playstation 3 platforms. The C4 engine was used to develop the Alligator project.

Controller – A C4 class for anything in the game that needs to be updated every frame, via its Move() function. Controllers are often paired with *entities* to create animated, moveable characters.

Entity – A C4 class for an animated model, usually paired with a controller. Our entity subsystem uses both controllers and entities to create all of the controllable and animated characters.

Flowfield – Flowfields are a steering technique of using force vectors to direct movement around obstacles and other obstacles. The Alligator project used flowfields for AIShip obstacle avoidance, and for water current simulation. Texture maps were used for the flowfields, with each RGB color representing a two-dimensional physics force vector.

OpenSteer – OpenSteer is an open-source steering library for artificial intelligence characters. OpenSteer was used for the AIShipController movement and steering.