Project Number: RL1-ALI1

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

Technical Development Manual

Worcester Polytechnic Institute

by

Dana Asplund

Kalun Fu

Yilmaz Kiymaz

Timothy Loughlin

January 7, 2007

Table of Contents

Table of Figures

1 Introduction

This document describes various technical aspects of interest to those pursuing further development to the Alligator game project. For a more general overview of the complete project, please review the *The Alligator: A Video Game History of a Civil War Submarine* project report for the Major Qualifying Project. Many aspects of the design, content creation, the physics system, and the artificial intelligence system are described in detail in the project report.

This technical manual provides additional information that we felt is of interest, but of too much detail for the project report. We've included as much information that time allowed to this document. For further resources, please visit the C4 Engine wiki, API, and forums at:

http://www.terathon.com/c4engine/developer.php

2 Functional Design

This section outlines various aspects of the functional design of the game, pertaining to the class structures and important function calls.

2.1 Developer Console Commands

The following is a list of the custom console commands implemented in the game:

- "load *name*" – Loads the world indicated by *name*.
- "unload" – Unloads the current world.
- "physics" – Toggles physics updates. This will halt all movement in the game.
- "bounds" – Toggles physics bounding volume rendering.
- "currents" – Toggles physics water current forces.
- "ai" – Toggles artificial intelligence updates.
- "camera" – Toggles camera effects such as shake and sway.
- "weather" – Toggles rain and snow effects.
- "fog" – Toggles underwater and above water fog effects.
- "effects" – Toggles all other particle effects.
- "freemove" – Toggles developer movement controls on/off.
- "time" – Cycles through normal game time, fast, faster, and realtime.

2.2 Game Runtime Sequence

The entire application is begun with the Game class constructor. This class manages the entire game flow, and stores game state flags and settings. The constructor initializes all of this data, including registering all exposed resources to the C4 World Editor. If objects such as custom markers, entities, controllers, and particle effects are registered, they will be assignable within the World Editor without recompiling code (providing they have the necessary code within their own class definitions).

The game then starts the interface system, opening up the MainWindow class for the main title screen. All of the interfaces and windows have global Singleton reference pointers, such as TheMainWindow for the MainWindow class. Thus, clicking on the

“Credits” button on the MainWindow will close that window, and open the CreditsWindow class via the TheCreditsWindow pointer.

Starting a mission involves opening the MissionPlanningWindow. If the resources exist for mission planning, detailed later in this document, then this window is created, otherwise the level is loaded directly from Game::StartSinglePlayerGame() function. The mission planning code reads and writes the configuration for that world, saved in the C4/Data/Engine directory. This includes the various slider values, which are used to set the weather and time conditions for the level, via the weather particles and EnvironmentController class.

The GameWorld class stores the game level from the world file, as well as the cameras for the game. It keeps a current camera pointer name playerCamera that is changed to point to the chosen view during the game, by calling the GameWorld::ChangePlayerCamera() function. This class also collects markers in the zone for the spectator camera list, crew entity locator list, and spawn point locator list. The world is often referenced in code to get zone references, the player, and other information.

Once the game world is loaded, the physics world and OpenSteer worlds are built, and the Game::BuildGameWorld() function is called to set water levels and territory controllers. The Game::InitEnvironment() function creates and initializes the fog, rain, and snow systems.

Entities are created through a factory method called Game::CreateEntity(). This creates the appropriate controller class, fetches the entity model, and assigns the controller to the entity. It also assigns physics if necessary, and then adds the entity into the appropriate zone through the zones Node::AddSubNode() function.

After the entities from the world file are created, the player and crew are spawned, the DisplayInterface class is created, and the game enters the GameWorld::Render() loop

where physics, AI, fog, and weather are updated, and the world is rendered each frame. All other updates in the game are handed by Controller::Move() functions in the entities and other controllers. Check these functions first, along with the constructor and Controller::Preprocess() functions, for details on specific custom controllers.

2.3 GameData

The GameData class stores the custom mission planning settings for the game, including water height, weather intensities, cloud levels, and time. Its data is loaded first from the configuration file, and any changes made to the sliders in mission planning will update its data instantly. This is frequently referred to through its TheGameData global reference pointer.

2.4 Avoiding Circular Dependencies

A frustrating issue we encountered during development was that of circular dependencies, including files via the #include tag that included the file in question, creating a cycle. When this happens, you frequently get a VisualStudio along the lines of "No default constructor available" for the class. The solution is often to use a forward declaration in the header file (H), where you only need to declare a variable of the desired class. For example, you want to include "MyClass.h" in your file to add a line like:

MyClass myClass;

If this causes a cycle and error like that mentioned above, you would remove the "#include Myclass.h" line and add line after the includes of:

class MyClass;

Then, add the "#include Myclass.h" to your source file (CPP), and it will compile and run. You have removed the cycle, and can use things such as myClass->function() now in your source file.

2.5 Pack and Unpack

Serialization for saving/loading games, and world editor settings, are handled through the Packable::Pack() and Packable::Unpack() commands for all controllers, properties, and anything else that uses Packable. This is necessary for storing World Editor settings and registering classes in the World Editor.

When changing these functions, you will corrupt your world files if you do not perform the proper steps. Since worlds must be unpacked when opening the file, and you add new pack and unpack code, the data has not been initially packed, and it will crash. Therefore, you can do two things to avoid the issue.

First, you may comment out all unpack code, compile, run, and save all worlds, thus packing your new information in. Then quit, uncomment the unpack code, compile, run, and save all worlds again. If you compile all code and run without doing this, you will corrupt any world you open.

Second, you may open up all worlds that contain the controller or other code that you are editing, before you compile the new code, and remove all of the references to that code. For instance, we often changed the PhysicsBodyController and GameCharacterController pack and unpack. Knowing we were going to working on them, we would first open the worlds, remove all physics and entity controllers, save, and then write the code. Testing would be performed on a single disposable level. When completed, simply open your worlds again and re-apply the controllers, save, and all will function with the new code.

3 Mission Planning

Every mission is made up of many different grids and GUI elements. A mission is made up of:

- Map image
- Water chart
- Depth chart
- Depth darkener
- Current chart
- Key chart

All of these elements are image files, in the format of 32 bit TGA (32 bit for alpha transparency). Each mission has one of each of the previously listed elements, and they must all be in the appropriate folder (C4/Data/Gator/texture/Interface/*missionname*). If they are not there, mission planning will not appear for that level, and no configuration file will be written or read.

3.1 Map

The map file is an image of the mission area, in full detail. This is rendered directly, with no interpretation of the image data, but simply rendered to the screen (see current chart for an opposite example). The map should be only the water, the land, and any landmarks that are built into the level. In essence, the map file is a 2D representation of the geometry for the level. It also has to have on it icons for the landmarks, whether they be lighthouses, sunken ships, towers, or whatever. These are overlayed on the map for the players to see.

3.1.1 Creation

To create the map image file, just create a regular image, probably a screenshot from a camera looking down on the level geometry. The map is held in an ImageMapElement. This is due to the fact that the ImageMapElement creates a clickable image (like an image button), and gets the point at which the image was clicked. When the map is

clicked on, it gets the point at which it was clicked and sends that data to mission planning, where a waypoint will be placed at that exact location.

3.1.2 Naming Conventions

This has to be named “map”.

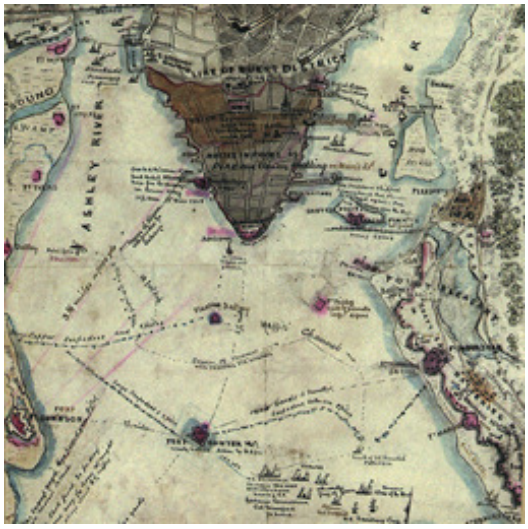

Figure 1 - Map Example

3.2 Water Chart

The water chart represents a mask of the map to show only where water is on the map. The players will never see this chart but it is used to determine if the player clicked on water or land, and if they clicked on water then a waypoint will be placed. The water chart must be a black and white file, with black representing water and white representing land. All values in-between are ignored.

3.2.1 Creation

To create the water chart image file, you should take the map and hand-paint all the land white (RGB value of 255, 255, 255) and all the water black (RGB value of 0, 0, 0). This can be rough, it doesn’t have to be fully accurate, but it should be close enough that major landforms aren’t cutoff. Finally, you should make all the white (all the land) to have an Alpha channel. You can do this by using the magic wand in Photoshop CS2 to

select all the white areas (holding Shift to select disconnected white areas), then do Select->Inverse to get all the black land areas. Go into the Channels tab of the Layers window (Window->Layers), and click on Save Selection as Channel. This will create an Alpha channel. Now, make sure that the alpha channel is visible (eye in the visibility box) and save as 32-bit Targa file with Alpha layers.

3.2.2 Naming Conventions

This has to be named “water”

Figure 2 - Water Example

3.3 Depth Chart

The depth chart represents the depth of the water at base water levels (no flooding or unusual currents taking place). The depth chart is an image file based off of the water chart, where only the water areas have depth values and land values do not. Depth values are created by using the blue values in the image. The depth chart is actually an overlay element also, meaning it will be drawn over the map image. Therefore, all areas that do not have depth (areas not blue) must have a black alpha mask layer applied to them to keep them from showing up in the GUI display. Also note that all the land must be black.

3.3.1 Creation

To create the depth charts, you should take the water chart and invert the colors, so that all the black turn white and all the white turn black. Now paint the water (the white areas) to different shades of blue, with black being the deepest and light blue (note that white is not the shallowest because in all the water areas the red and green component should be set to 0) being the shallowest. All red and green values have no effect on the depth.

Note that the game terrain will probably be generated using heightmaps. Heightmaps range from 0 to 1 (or 0 to 255). But when heightmaps are generated these values are scaled by a given factor. For instance, we could scale the heightmap by 10, so all white areas are at a height of 10 and 0 is at a height of 0. The colors on the depth grid have to be scaled by this same factor. The (0,0) pixel is the scale factor pixel. The red value of this pixel determines the scale of the heightmap. The other values are ignored, or the blue could be set if you want to use this as a water area.

The first pixel (0,0), is also used for the scale of the map. The heightmap scale is for vertical scale, but this scale is for horizontal scale. This means that 1 pixel equals something. For instance, 1 pixel may equal 1 meter, or 10 meters. Of course, there is an upper limit to the map scale because the green value can only go up to 255, but having 255 meters per pixel seems enough. Therefore, just set the green value to the pixel per meter that you want.

3.3.2 Naming Conventions

This has to be named “depth”.

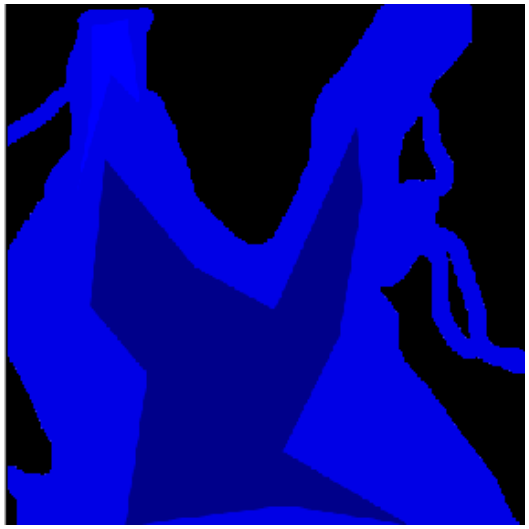

Figure 3 - Depth Example

3.4 Current chart

The current chart is probably the most complicated chart of all. All the colors in this chart are very important, the colors for the flow and even the colors for land. The current chart represents the current (tidal) flow for the map. The current is a vector, showing magnitude and direction of the current. All the color values in the chart represent a vector.

The RGB color for every pixel of the current chart is turned into a vector (x, y, z). However, only the G and B (green and blue) color values are used to show the current, so only the y and z of the vector (g=y, b=z) are used for the current. The y value (green) for the pixel represents how fast the current is flowing left and right. The z value (blue) represents how fast the current is flowing up and down. In Photoshop, colors range from 0 to 255. If the G or B value is between 0 and 128, then it is negative flow (either left or down). If the value is between 129 and 255, then the flow is positive (either right or up). For instance, the color value GB (255, 255) represents a flow that is very strong to the upper right. The color value GB (0, 129) represents a flow that is strong to the left and very slightly moving upwards.

To convert the color values to vectors we take the 0 to 255 range and break it into a -1 to 1 range, with -1 being 0 and 1 being 255 (the color value GB (0, 255) is equal to the vector (-1, 1)). With this, 128 becomes the zero point (so the color value GB(128, 128) is equal to the vector (0, 0), showing no flow).

The current grid is also used in the Mission Planning window. Every pixel in the current image should have a red value of 0. However, Mission Planning goes through the current image and finds all pixels that have a red value of 255 (full red) and places an arrow icon element to show the flow of the current at that location on the map. When a pixel does not have a red value of 0, and it has a value of 255, and this pixel is flagged for an arrow icon to show the flow to the players.

The color white in all the previous charts showed land, but in this chart land has to be maroon (128, 0, 0). Do not brush large amounts of white, this will cause many arrow icons to be drawn (white = (255,255,255), meaning it has a flag value of 255, so it signals Mission Planning to draw arrow icons).

3.4.1 Creation

In order to create the current flow, which is the hardest image to create, you should start with the water chart again. First, make sure there is no white in the image. Turn all the white to maroon (128, 0, 0) and make sure the rest is solid black. Use the brush tool and the color sliders in the color window (Window->Color) to draw the current flows. You should not be choosing colors and drawing colors on the image (unless, somehow, you know the green and blue values for the color, which seems impossible). Instead, you should set your red values to 0, and choose your green and blue based on the flow you want. Remember, green values are flow from left to right, and blue is flow up and down. Also remember that values greater than 128 are positive values and values below 128 are negative (Up and right are positive. Down and left are negative). Keep in mind that great values (0 or 255) are very fast moving flows. If you want a fast moving current in the direction of northeast you would set the red to 0, the green to 255, and the blue to 255. If

you want a slow moving flow directly south, you would set red to 0, green to 128, and blue to 90.

Now, after you have all the current flows set up, you should go back through and make flags, setting the red values to 255 so that arrows show up in the MissionPlanning window. To do this, create another layer above the layer you drew the currents on. In the Layers tab of the Layers window in Photoshop (Windows->Layers), set the new layer's blend mode to Difference (the other layer's should be set to Normal). With that Difference layer selected, use the Pencil tool, set the color to full red (255, 0, 0), and click everywhere you want an arrow icon to show up. What this difference layer is doing is just adding the 255 red flag to the pixel you are clicking on.

3.4.2 Naming Conventions

This has to be named "current".

4 World Creation and Editing

Terrain generation and model creation have already been outlined in the project report, under the Art Assets section. We will therefore provide several important steps in the creation and editing of any levels to be used by the Alligator game. To view a working level, refer to the Norfolk world. You may open the file, copy things such as the water or sky group nodes, and paste into a new world you are creating.

4.1 Zones

All worlds start with an infinite zone. This contains all subzones, and the skybox for the world. You should apply a Bullet Zone Physics property to the infinite zone in the Node Info window, usually with a -10 setting for the Z value for gravity. Otherwise, physics will be disabled for the level.

We separated the main game world from the interior view of the submarine through the use of two box zones. This occludes the interior from being rendered when in the exterior zone, and vice versa.

4.2 Water

In order for water to function correctly in the game, you must create a Fluid plane with the water material setup as in the Norfolk level, and an underwater Fog layer. These nodes must be grouped together in the World Editor, and the group node must be named "WaterNode". The z location of this node should be entered into the configuration file for that level with the initialWaterHeight variable. The fluid plane should be at a relative z position of 0, and the fog should be slightly below this, such as a z value of 0.005.

4.3 Sky

For proper sky functionality, refer again to the Norfolk level. First, place a skybox node in the infinite zone. Next, all of the sky nodes (clouds, sun, moon, and stars) must be grouped under a node named "SkyNode", and the Environment controller must be applied to it. Within this node, there must be a dome named "Clouds" for the cloud layer,

with Render in Effects Pass and Render as Decal selected. A “Sun” group and “Moon” group must also be present, each containing an infinite light and any flare effects desired. Also within the SkyNode, a starfield particle effect must be placed, and named “Stars”.

Upon world loading, the EnvironmentController is created, and the EnvironmentController::Preprocess() function traverses the scene graph for a skybox node, and the SkyNode and its children. These are initialized according to the Mission Planning sliders. The EnvironmentController rotates the sun and moon, animates the zone’s ambient light levels, enables the stars at night, swaps skybox textures for different times, and swaps cloud textures according to the clouds slider setting. If these nodes do not exist, and are not named and grouped properly, the EnvironmentController will not function.

4.4 Terrain

After terrain is imported according to the project report specifications, you should apply a Bullet Uncontrolled Rigid Body property in the Node Info window, and set its collision type to “Heightfield”. This will enable physics collisions with the terrain.

4.5 Entities

Entities are placed via the entities page in the World Editor. Note that the red axis arrow (local +X) indicates the orientation direction of the entity. Once they are placed in the world, you can change their physics properties in the Controllers tab of the Node Info window.

4.6 Markers

Locator markers are used for many things in the game. A locator of type “spwn” indicates spawn points, “spec” indicates spectator camera points, and “crew” indicates crew entity locations. Additionally, “weap” indicates weapon positions within an entity model (MDL) file, and “camr” indicates the 1st person view camera placement for the submarine’s entity model (MDL) file.

Reference markers are used with any world file saved in the C4/Data/Gator/world/ref directory. These markers create a single instance of the world file placed at that location, and all other reference markers of the same type refer to this instance. Simply place a reference marker where you want, and choose its Referenced World Name in the Reference panel of the Node Info window. These rapidly speed up level design and performance. We highly recommend you save all geometry, including those used for model (MDL) files, as reference world files.

5 Physics

This section describes how to use the physics system in Alligator, such as how to give objects physical models and place them in the simulation.

5.1 Sub Surfacing and Diving

The way that the sub surfaces and dives is by adjusting how much water it has in its ballast tanks. In essence, this changes the volume of sub when it gets submerged, which then changes the buoyancy of the sub.

Every model in the physics world has a body size, stored as a vector (Vector3D size). The volume of any given physics body, such as the sub, is simply:

size.x * size.y * size.z

The way buoyancy works is that a force is applied for the volume of water displaced by a given body. Here is the formula that is used for buoyancy:

water_density * volume * gravity_acc

This is known as the Archimedes' Principle. For Alligator, the equation changes only slightly. Typically, water density is 1000 kilograms per cubic meter. However, by using this water density, the mass of everything getting applied buoyancy would have to be ridiculously high. Think of a beach ball. It has a lot of volume but very little mass, so when it gets submerged it wants to surface very quickly. If this density was used, the mass of our sub and ships would have to be close to 30,000 kilograms or more, because the volumes of the ship's hull and sub's interior are decently large. To scale this down to a manageable level, we used a water density of 1.0.

There is another factor in our equation, the currentBuoyancy factor, stored as a percentage, which represents the ballast tanks for the submarine. Ballast tanks decrease

the volume of the body when filled with water, so currentBuoyancy is a percentage of the volume that is being applied in the buoyancy. If currentBuoyancy is 1.0, then all of the volume is being used for buoyancy force. If currentBuoyancy is 0.5, then only half of the volume is being used for buoyancy forces, which means that the ballast tanks are filled, and thus they take up half the sub's volume. So the equation we really use for buoyancy is this:

water_density * volume * gravity_acc * currentBuoyancy

Each GameCharacterController can assign a starting buoyancy, a minimum buoyancy, and a maximum buoyancy. These are all percentages that affect currentBuoyancy. They are limits to how filled the ballast tanks can be. For the sub to remain still in the water, without rising or sinking, there needs to be a force that counter-acts the force of gravity. This can be calculated using our buoyancy equation:

water_density * volume * gravity_acc * currentBuoyancy = mass * gravity_acc

To find the neutral buoyancy force, we simply solve for currentBuoyancy. All of the variables in the equation are known except for currentBuoyancy. Gravity cancels out, and the water density is 1.0, so the neutral buoyancy percentage is:

Mass / volume = neutralPercentage

It is interesting to note that you can go in the reverse direction as well. When you are deciding what to set the mass of a body to, you can use the reverse equation and set what value you want the neutral percentage to be:

neutralPercentage * volume = mass

Typically, you should use 0.5 because this is right in the middle of the 0 to 1 range, making it easy to have full control of the buoyancy if you want to give the body a surfacing ability.

The sub surfaces and dives through the crew. The crew manipulate the sub's currentBuoyancy to cause it to surface or dive by calling the sub's PumpBallast function:

PumpBallast (float perc);

The perc variable is what percentage the crew is trying to set currentBuoyancy to. This function either floods the ballast with water; if the currentBuoyancy is greater than the desired percentage, or if the currentBuoyancy is less than desired percentage. The rate at which the tanks are filled is determined by some of the sub's variables such as ballastFillRate and ballastFloodRate. The crew continues to pump the ballasts with air or water until they reach the desired fill percentage.

You can use a simple formula to change the rate at which the tanks are flooded or filled:

(100 / x) / 100,
where x is the time in seconds you want it to take.

This equation returns a float of the percentage rate per second that the tanks are filled or flooded.

6 OpenSteer

6.1 Build and Setup of OpenSteer

Connecting OpenSteer with C4 can be tricky, because it uses different libraries. Some of them are conflicting with C4. For example, C4 has its own memory management system, which creates a problem when we try to use the standard "new".

There are a few things we have done in order make C4 compatible with OpenSteer in VisualStudio:

- Add opensteerd.lib as additional dependencies.
- Include OpenSteer header files directories in both the Linker and General tabs. You will need to manually delete the OpenGL code first.
- Add Libcmtd.lib to Linker/Input/Ignore Specific Library.
- Include the OpenSteer lib file in the project settings.
- Add a macro in the OpenSteer header files so that it won't complain about "new" has already be defined.

An easier way is to use the pre-built version of the opensteerd.lib file that comes with our code, and the existing Alligator solution for VisualStudio.

7 Suggestions

As detailed in the project report, we have offered several suggestions for future development of the title. The report contains detailed descriptions of additional features and missions, and the desired goals of the product for educational purposes. The following is a short list of specific things we have identified relating to technical development:

- If Macintosh compatibility is no longer required, remove Bullet and use PhysX for physics. PhysX is much more stable, robust, and feature rich. The integration with C4 is constantly updated for each build.
- Remove OpenSteer and write a custom AI engine with steering and aiming algorithms. OpenSteer proved to be general and complex for the needs of the game.
- Implement Save and Load functionality. This will require adding many variables to the Pack and Unpack code for entities, the environment, and other classes. Once this is performed, it should be a trivial implementation.
- Add more content, such as levels, the other weapons, enemies, and obstacles.
- Add networked play (using networked PhysX), and implement player controlled ships, humans, and gun batteries.
- Enable players to be a base human, and the ability to take over and pilot vehicles and weapons.
- Add more particle effects, and refine the existing particle effects.
- The upcoming “Fireball” series of releases for C4 will totally change terrain, weather, water, sky, and foliage generation and implementation. Keep up-to-date on the C4 engine progress.

8 Conclusion

We hope this document has aided your understanding and development of the Alligator project. We took much pleasure in the development of the prototype, and hope to see its continued development. We feel we have provided an adequate framework on which to develop the title, even though some areas need to be improved or rewritten. Again, we highly recommend that you utilize the wonderful community of the C4 forums for additional help, and refer to the project report, project code, and C4 engine source code.