Game Design Group Project

Design Document by Elijah Houle, Gregory Norton, and Bo Wang

Overview

The idea for this game takes the real-time strategy elements from games like Warcraft/Starcraft and puts them into an underwater environment. Submarines take the place of both transport and militarized ships, with different types of divers as workers and soldiers. For this project, the game will be multiplayer-only, with two clients and a centralized server.

Each side has a central base, the objective being to destroy the other side's. Players are given worker units at a somewhat randomized rate, with the option to use them to mine for resources, build factories using the mined resources, or upgrade them to soldiers. Some factories build submarines as transport vessels, but the submarines can be upgraded to military vehicles at a cost of resources. Note that the low bar has a minimal tech tree with only a single upgrade for each unit. All actions take some variable amount of time depending on the task.

The user interacts with entities similarly to many other RTS games: by left-clicking the entity to control (or selecting a group of entities within a rectangular region by holding down the mouse from the top-left to the bottom-right of the desired region). Right-clicking by default will either move to a bare spot of ground, mine if a resource area is targeted, or attack if an enemy is targeted. For workers, the interface toward the bottom of the screen displays alternative actions to perform for building different types of factories. The low bar implementation of this project would only include a single type of factory for building submarines.

Submarine TITANS. [http://i1-games.softpedia-static.com/screenshots/Submarine-Titans-Patch_2.jpg]

The only game we found that predates this idea in the underwater RTS genre is “Submarine TITANS” published in 2000. Its Wikipedia article claims it to be “the only game in the RTS genre where the entire realm takes place fully underwater” (http://en.wikipedia.org/wiki/Submarine_TITANS), a distinction that could be lost after this project is completed.

Development Strategy

This project will continue to use the JIG libraries for the main game code. Some basic rendering code might be reused, such as entity highlighting and health/progress bars from Elijah’s “Reap the Harvest”, as well as elements of Dijkstra’s and collision detection from both Greg’s “Smash TV Season 2.0” and Bo’s “Battle City”.

Until recently, the plan to handle the movement logic for the terrain map used in the game was to develop a custom de-serialization library based on XML or JSON as a technical showpiece, but after discussing the schedule with the instructor, the development team decided to go with the Tiled tool for terrain map development targeting an isometric projection. Tiled is a generic tool for tile map editing and has support in Slick2D via the TiledMap class.[1]

Sticking Points

One of the sticking points, described in more detail in “Real-Time Multiplayer Networking” under “Technical Showpieces” with the problem of desynchronization, will be coordination between the two clients. Game processing would need to lock itself at a standard frame rate and compensate for processing speed differences between machines.

Another challenge in technical showpieces will be developing a sufficiently fast implementation of Dijkstra's algorithm for a given map size since at least one path-finding calculation will be necessary for each entity following movement orders, and movement around obstacles will require as many recalculations per second as possible. A combination of increased concurrency and adjustment of map node count should result in reasonable movement calculation time that does not affect other aspects of the game, but finding a playable balance will be a trial-and-error process at first.

The game's use of Isometric projection will require careful consideration of "world coordinates" against "screen coordinates" when rendering each player's view of the game in progress. Rendering is also complicated by the "fog of war", limiting the visible details on each side to what is "seen" within a specific radius of the players' respective units.

Creating sufficient artwork for the individual units in order to present a reasonable Isomorphic projection could be a challenge in the given time frame, and this is one reason that the "low bar" provides a very limited "tech tree".

[1] http://slick.ninjacave.com/javadoc/org/newdawn/slick/tiled/TiledMap.html

Milestones

1. Identical map is shown on both clients, with a single simple worker for each player. Each player clicks a position to move their worker, and the movements are coordinated such that each client displays the same result. This demonstrates the basic networking requirement.
2. Introduce fog of war, in which each player can only see the portion of the map that their units have discovered.
3. Introduce central base to each player, mining operation, and new upgrades to complete low bar.

Roles

Although the entire group coordinates design and may share section development at different times, each group member manages a section of the project:

- Elijah – Interface (entity art, movement, collision)
- Greg – Object format development, Parallel Dijkstra implementation
- Bo – Networking architecture and protocol

Technical Showpieces

Real-Time Multiplayer Networking

The first technical showpiece will be the multiplayer networking. The networking model we will use is the client/server lock step model, which is widely used in RTS games[2]. Each client sends their command to the server. The server can either 1) simply send all commands to each client, and let everyone run everyone's commands locally, or 2) perform all the computation and send the result, and the client is just a terminal which only performs rendering. The basic idea of lock step is to divide the game into a series of turns. In each turn, every client executes the exact same code at the exact same frame rate.

There are two problems involved in this architecture. The first is determinism. Since each player is independently updating the game state, it means the game simulation must be fully deterministic. A tiny desynchronization will become further apart as game goes on. Fixing a desync bug is usually a tough task.

The second problem is latency. To execute the same commands at the same time, every client must wait for the slowest player. For example, player 1 has latency of 100ms, player 2 has latency of 200ms. Ideally if player1 sends a command he will be able to process the command at 100ms if he is the only player. However, if player2 joins the game, both player1 and player2 have to wait 200ms each time in order to be synchronized. During this period of time the game is forced to stall.

[2] http://www.altdevblogaday.com/2011/07/09/synchronous-rts-engines-and-a-tale-of-desyncs/

Parallel Dijkstra's

The intent behind this technical showpiece is an attempt to improve the AI behind game entity movement by implementing a parallelized Dijkstra's Algorithm engine, the "Parallel Twist" as described in the "Path Finding" lecture notes for the course.

JSR 166, a feature set included in the Java 1.7 release, adds concurrency utilities to Java.[3] These are available to Java 1.6 developers as part of the "jsr166y" package which is available from

http://g.oswego.edu/dl/concurrency-interest/

The "Fork/Join" framework from the package will be utilized in an attempt to implement the parallel version of the algorithm as outlined in the lecture notes. The procedure to use the framework is to allocate a global "pool" of tasks that will execute concurrent processes embedded in objects sub-classed from the abstract task type. A task enters into the pool via the "fork" call, and the parent process collects the result through the "join" method.

Contrived examples from online tutorials such as the one at JavaCodeGeeks[4] demonstrate the basic use of the framework while conceding that performance may be better using solutions that are more traditional. In the case of Dijkstra's Algorithm, starting and stopping threads impose a performance penalty that make the single-threaded approach more efficient for graphs smaller than a specific size, and part of this implementation effort will be to discover, in general terms, this inflection point for CPUs with different core counts.

The specific application outlined in this design document will eventually require multiple path finding recalculations with every frame refresh once moving obstacles are introduced which some of the entities must avoid. Through parallelization, the development team expects to significantly increase the size of the graphs that can be recalculated in the allotted time frame while maintaining the 60 frames-per-second refresh rate.

Initial effort with parallelization will focus on the section of Dijkstra's Algorithm that extracts the minimum cost node when laying out the path, but other areas of the algorithm such as establishment of lists of adjacencies in the graph data structure may benefit as well.

To establish an idea of what size map can be handled more efficiently under parallelization of Dijkstra's Algorithm, the implementation will include JUnit-based test code that allows different size graphs to be created and tested quickly without running the full game. An acceptable running time will allow the path finding calculation to run for each moving unit while leaving CPU time for other necessary calculations.

---

[3] http://www.jcp.org/en/jsr/detail?id=166

[4] http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html

High Bar

The high bar adds an oxygen restriction to the game's strategy, with another type of factory that extracts oxygen from seawater. Divers (workers and soldiers) would need to stay within a certain radius of such a factory or a submarine (which generates its own oxygen).

The high bar will also add more details on control and user interface to make the game more like a real RTS game. For example, when select a unit, its status should be displayed on bottom of screen. When select a group, player should be able to switch between units in the group to view their status. Player can either navigate the map by clicking on the small map on the bottom or scroll on the main view.

More AI on the units. For example, a unit should auto attack an enemy if they are close to each other. A high range warrior should attack from a distance instead of move towards the target when it receives an attack command.

In addition, the highest bar offers several more types of factories, units, vehicles, and AI sea creatures. The bar increases in complexity with each attribute added to the tech tree.