Team MaVeN

Created by: Kevin Johnstone, Joshua Zeder, Farzon Lotfi, and Kyle Davis

Game AI Final Project

How to run:

In order to run World War Farzon, import the wwf project into an IDE and run the Game.java file. If running in a non linux environment, you have to change both lwjgl native library locations. To do this in Eclipse IDE right click the wwf project folder and go to the libraries tab. On both lwjgl.jar and lwjgl_util.jar expand the menu options and click on the native library location and select edit. In the path change where it says linux to windows or macosx if running macosx.

Game Description:

Our game, World War Farzon, is a tug-of-war style game where a human player competes with an AI to be the first to destroy the opponents Nexus. In order to accomplish this, players build buildings that spawn units corresponding to each building type. Each building has a specific mineral cost, build time, health points, and unit spawned. Once a building is built, the building will continue to spawn units at a building specific rate for no further investment in minerals. The unit spawned corresponds to the specific building type and have various traits such as build time, armor type, health points, movement speed, attack range, attack speed, and damage. Each unit has a damage multiplier for how much damage is done to another unit depending on the target units armor type. An example is a marine has a damage multiplier of 0.5 against heavily armored units causing the attack damage to be cut in half. This allows for more dynamic unit compositions as units are not simply better than the rest, they all have weaknesses which can be exploited by building the proper unit composition.

Visualizer Control:

Use the arrow keys to move around the map. The ‘m’ key causes the minimap to toggle between visible and hidden. the mouse scroll allows you to zoom in and out.

How to Play:

Players can switch building types, which can be seen in the middle with the mineral cost above the building type, using the number keys 1-5. In order to build a building, click on the left side of the map after selecting the proper building type and if you have enough minerals that type of building will appear and begin building units. Units will spawn and attempt to travel down the lane closest to the building and will engage in any enemy targets encountered. For every enemy unit you kill, you receive one additional mineral and vice versa. The building is a one time investment where all units are built for free once you buy the building. The goal of the game is to destroy the opponents nexus located in the center of the base on the right hand side. Whoever destroys the other’s nexus first is declared the victor.

The following is a list of building descriptions denoted by the number key associated with that specific building. Building one is a barracks that produces marines, two is a dojo that produces heavily armored but short ranged units, three is a ghost academy that builds snipers with a long, powerful attack but slow attack speed. Building four is an rpg factory that builds a long range unit that acts as a more powerful sniper but with greater hp and range than the traditional sniper. Building five is the tank factory produces a very powerful but expensive tank that is heavily armored with a high hp and strong attack, but the time to build a tank is long.

Goals:

The goal of the project was to create a tug-of-war style game where a player must compete against a computer opponent that has a high level of intelligence. We needed to be able to create two separate artificial intelligences. The first AI controls the movement of units as well as the combat between them. The second AI plays the game with the same set of rules as the human opponent and must be able to make decisions about what types of units to build, where to place its buildings, when to spend its money, and timing its building placement to sync the creation of units with previously placed buildings.

The Game Engine:

The game engine is built on top of lwjgl. As the name implies it is written in openGL using JNI for native callbacks.

Map Generation

We utilize a three dimensional array where the first two dimensions corresponds to the x, y coordinates of the tiles that make up the map with the third dimension allowing layering of the map. We have a terrain layer dimension in which we store the types of terrain including ground, walls, and water. The map generation is done in three passes on the array. The first pass generates the water. The second pass generates the walls, and the third pass will set all unset values in the array to ground. This technique enhances performance by reducing the number of draw commands on the display. Once the terrain has been generated we move to the building layer of the array and add all the buildings before moving to the final layer of the array, the unit layer.

Software Design Patterns

GameMap.java utilizes the singleton design pattern. We utilized this pattern because only one instance is ever needed. Having more than one GameMap could only lead to errors

and bugs since GameMap is responsible for a large portion of the game logic.

Different types of Buildings (BuildingImpl.java) and Units (UnitImpl.java) are created using the common factory and builder patterns. Units and Buildings share many of the same attributes so we used a builder class to make it easy to rapidly create and change units with minimal coding overhead. You can see in the factory classes how adding a new unit to the game is done in very few lines of code by just specifying certain variables in the builder that should differ from the defaults.

OpenGL Draw Hierarchy

Everything is drawn using an abstraction of sprites, sprites are basically just squares with textures on them.and a color association from the player. At the moment OpenGL 1.1 is being used. This does not provide optimal performance, but it is not an issue because we intelligently choose when and how we draw the sprites so that overlapping sprites don't waste hardware resources.

We have a menu system that is dynamically placed on the map. The menu holds game data including the players current income, current minerals, current building type selected to build, and a minimap. The menu visibility can be toggled by pressing the 'm' key.

We implemented the ability for scrolling and zooming on the game map. We implemented this by taking advantage of the Opengl viewport. This allows a user to zoom to their preferred viewing perspective and moving as implied allows the users to move their viewpoint around the map.

Artificial Intelligence:

World War Farzon required the creation of two separate artificial intelligences. The first AI handles the movement and combat between units. The game has two lanes of where units

travel separated by walls and water in the center of the map. We needed to be able to have units spawned by buildings attempt to move to the lane that is closest to said building. In addition we needed units to attempt to engage in combat if within a reasonable distance. The end goal of all units is to destroy the enemy nexus.

Path finding was an interesting challenge because units move by pixel specified by the unit movement speed as opposed to moving by the length of a tile each time. The game map is built of 32 x 32 pixel tiles that correspond to different types of terrain and overlapping unit sprites. In order to detect whether it was legal to move to a certain position on the map we needed to do collision detection on the unit sprite image and unpassable terrain sprites such as water given the top left hand corner coordinates of both. We originally considered performing an A* graph search for the AI. However when done by pixel these leads to such a large amount of expanded nodes that the run time was too slow. We needed to come up with a path finding algorithm that could account for a large amount of units with the ability to account for unpassable terrain.

The solution we chose to implement was utilizing influence maps. In the general case Influence maps work by assigning a value to each cell in a partitioned game map where the cell corresponds to node or tile. These values are assigned based on game factors such as what objects are contained within a tile. The values are spread across the rest of the cells through a process called diffusion. Diffusion works for each cell by taking the average value of the cells to the left, top, right, and bottom and placing the result in the cell. The implementation is very game dependent.

Our implementation of influence maps involved the creation of four two dimensional arrays where each index of the array corresponds to a tile. Each player has a diffusion array as well as a corresponding diffusion image array. The reason we chose to use a second array is

we found it helped to spread the values more evenly. If we were to only use one array for diffusing values, updating a tile's value in the diffusion array will have a direct effect on updating the value of the next tile in the array. By using a second array we are able to update the first diffusion array based on the values in the diffusion image array. We then diffuse the values from the diffusion image array back into the diffusion array using the values in the original diffusion array.

In order to achieve proper path finding behavior using diffusion we gave high values for enemy units and buildings encouraging friendly units to move towards the enemies. We gave values of 0 for unpassable terrain such as water and walls and did not allow values to diffuse through these types of terrain. This resulted in the diffusion values from the bottom lane having a very limited effect on the values in the top lane.

When choosing where a unit should move the AI examines the neighbor diffusion values in the tiles surrounding the unit. We found that we were able to use the tile that the top left pixel of the 24 by 24 pixel unit is contained in when making movement decisions.

In order to implement combat we use the unit attack range which is an integer value specifying the number of pixels the unit can attack from its current location. The current location is considered to be the top left pixel position of the unit and the euclidean distance is used to determine whether an enemy is in range. If we find a unit is in range we attempt to attack the target. If a unit is found to be in range, the unit will not attempt to move until that target is no longer a possibility. The reason for this decision is because units have an attack speed measured in milliseconds. Once a unit attacks a unit must wait the amount of milliseconds specified by the attack speed before being able to attack again. We did not want units to be attacking then moving while waiting to reload because this would lead to units passing each other in the lanes.

We built a three dimensional array where the first two dimensions of the array correspond to the x and y position of a tile while the third dimension allowed the map to be layered. By layering the map we can check for cases where a building is on a tile, a unit, or a combination of the two as well as which player the entities belong to.

The second AI handles all the tasks the player controls. The first and most important task is deciding what composition to build. The AI decides this based partially on what the player is building. Each unit in the game has a counter unit, which means that given a healthy composition of units there is a mix of everything. So the AI sees what buildings the player has built and then decides what to build after making a guess on what unit it will need to counter. It also takes into account the buildings it already has built. For example if the player has built 5 Barracks and the counter to barracks is the Dojo, then if the AI already has a reasonable number of Dojos built to deal with the Barracks,it will not be as likely to build a Dojo as it would have been if it had no Dojos. This keeps the AI from leaning too heavily towards one unit in the early game, which would cost it in the later stages. Another way the AI decides on what unit to build, is the perspective income. In the later stages of the game the players accumulate income much faster than in the beginning. So the later it is in the game and the more units the AI is killing, the more expensive buildings it is able to construct. The more expensive the building, the better the unit produced, so the AI leans towards saving some minerals for a better building in the late game, rather than building a bunch of inferior buildings. Another input variable being used in deciding what type of building to create is the length of time it takes to spawn a unit. Some of the buildings take significantly longer to spawn a unit than others. This is used as a deciding factor if the enemies are extremely close to the AI buildings and it needs a unit fast. Vice versa, if the AI units are pushing the player, it has time to wait for more powerful units to spawn and can build the slower producing buildings.

Also the AI has to decide in which lane and when to place the buildings. The AI decides to do this in a slightly similar manner to the way it decides what buildings to produce. It bases it on the player and on what it has built already. The AI has rules to try and keep the lanes between the player and AI even. So if the player builds the majority of the buildings on the bottom, the AI will place most of its buildings on the bottom as well. This keeps the player from overrunning one lane. A power level is associated with each building, which keeps buildings from being counted as the same weight in the problem of deciding where to build. For example, if a player places a Barracks on the top and an Academy on the bottom, the AI will consider the bottom to be the bigger threat and place a building on the bottom before constructing one on the top lane. It also takes into account the placements of the units in lane. If the AI units are pushing one lane but not pushing the other, it is more likely to place buildings on the lane that is struggling to push back the player.

The timing of placing a building is also an important part in the AI. Units clumped together is more effective than sending in units one at a time. Given this, it makes sense that the AI tries to place the building when the buildings around it are producing a unit. This allows for units with the same spawn time to group up for harder pushes. This makes a big difference later on, because larger groups of units accumulate and quickly pick off any stragglers.

Major Challenges:

A major challenge we faced when making the player AI, was deciding when to build and when to save our minerals to produce better units. The AI kept building cheap units when it received enough minerals. This was severely limiting the effectiveness of the AI, because if it did not win the game in the initial part, it would get obliterated mid to late game by much more advanced units. We solved this by doing a number of things. The first and easiest strategy to

introduce was deciding to save more minerals the later it progressed into the game. The second thing we added to deal with this problem was measuring how far the lanes were pushed in the computer's favor. If it was pushed in the computer;s favor, it is much more likely to save minerals and wait to create better units. However, if the lanes are pushed in the player's favor, the computer is more likely to spend minerals to try and regain lane control. We also added a projected minerals. So if the AI had been collecting minerals at a pretty steady rate that was only increases, we can make a fairly accurate guess as to future minerals. If the guess for future minerals was high, the AI would be likely to wait for an expensive building, and if it was low, the AI will spend the money on a lower cost unit. The last piece we added to deal with this difficult problem was a building queue with a timer. The building queue is kept small by removing the front of the queue every time it builds or in a set amount of time to avoid waiting indefinitely for an expensive building. All of these features working together in our computer AI effectively solved the difficult problem of deciding whether to spend the minerals on a cheaper unit or store the minerals for a more effective expensive unit.