Eashaan Kumar (ek485)
Nickolas Cavagnaro (nrc48)
Ryan Meredith (rcm269)

3110 Design Document: OCamlot

System Description

Core Vision

We will be creating a strategy game in which the player tries to conquer enemy structures in increasingly difficult maps.

Key Features

- Random Map Generator
- Tower Attack System
- Enemy AI
- Sprite Animations
- Sound engine

Narrative Description

The map starts with various towers, one of which is the player's base. From the base, the player must attempt to conquer all towers to beat the map. In order to do so, they must send their troops from the base tower to another tower. After conquering other towers, the user may control the flow of troops from multiple towers at the same time. As the player attempts to conquer enemy towers, a bot of our creation will attempt to conquer the player's towers. Troops are constantly regenerated, but the user must be resourceful lest they lose to the bot due to poor troop-positioning. As the game progresses, "attack towers" become available to the players to conquer. These towers shoot at the players if they come too close.

System Design

- engine.mli
    - This module will contain the main game loop and will be responsible for initializing the game state, updating the frame, updating all entities, and taking in user input.
- state.mli
    - This module describes the functions and data structures necessary to update the game state so that the engine may correctly infer the graphics that need to be

drawn and so that the AI can seamlessly transition from one state to another in checking what the non-player-character (NPC) will do next in its conquest to dominate the user.

- ai.mli
    - This module describes functions that will implement the opponent. This will include a function that takes in a state and an evaluation function and returns what it deems to be the optimal move for a given side. One of the options will be for the AI to pass since this game is real-time.
- sprite.mli
    - This module describes data structures that store images pertaining to each entity. It also describes functions that animate entities based on their set of sprite images.

Data

State.state

In state.ml we will store information about all of the towers in the map in records containing troop number, allegiance, location, and graphics information. We will also store records for movements which are used to determine where troops are moving to and how far along they are in their path. These will all be used to draw the game state.

AI

In ai.ml we will use a module to store an instance of an AI which will contain a heuristic evaluation function and method for using the function acquire the best move.

Engine.input

In engine.ml, we will use the input data structure to store key and mouse events. Then, input will be passed into Engine.update that will change the game's state using the input. It will contain a array of keys pressed in the last frame, keys released, a boolean for mouse pressed or released, mouse position and if the user interacted with on-screen elements such as selecting towers to send troops from and to.

Engine.properties

The properties stores map information such as number and location of towers, AI difficulty, etc. If it contains invalid information that prevents the creation of an initial game state, then an Engine.Init_Failure exception will be raised and Engine.init_game will fail.

Other Data Structures

We will be relying heavily on Hashtables and Hashsets for storing towers and troop information because of their O(1) get and insert times. This is useful if we want to fetch information about an entity based on its name or id. The hash function will map each id to the data and hence speed up the whole process significantly. We will use arrays instead of lists wherever we can, again because of their O(1) run time.

External Dependencies:

We will use the graphics library to render our game. It contains many helpful functions for rendering images to the screen and getting user input from the mouse and keyboard. We don't intend to rely on 3rd party libraries as the ones provided should be sufficient.

Testing Plan

Overview

To test our game, we will do interactive testing for each file when possible. For the GUI, we will pass in states that have known configurations and verify that the output image is what we expected. For file like ai.ml and state.ml we will want to test them in together. We will test this interactively by playing the game and making sure that no illegal moves are allowed, that legal moves are allowed, that the game starts and terminates at the proper states, and that gameplay is smooth. For the AI, we will simply play games against it to test its abilities and decide if it's a reasonably intelligent opponent. This is difficult to quantify, so we will have to judge it based on our own knowledge of the game and the results from our games.

Unit and Module Tests

It is likely that the only unit tests we will write are for the state file. For ai.ml and engine.ml the outputs are not something that one can quantify and so those not be tested in this manner. For state.ml we will want to write tests that input a move to a state and test it against the anticipated output state.

Team Commitment and Accountability

We'll examine github to make sure that everyone if contributing a substantial amount to the project. We will all be mostly working on our own parts of the project, so it will be easy to see who has done what. In addition, we will all be working until the project is done, so nobody can get out of doing work by quickly submitting sub-par work.