Jessica May

Joshua Linge

Assignment 1 Report

**Chosen Game Engine: Unity**

One of the reasons we chose Unity was because we have some prior experience with Unity and C#, and wanted to learn more about it. Unity provides many features and libraries that other game engines would require us to create ourselves. This makes Unity an easier engine to work with, and also less time consuming. For our game we needed a 2D engine, and Unity recently updated to provide necessary 2D framework. Unity has built in physics, collision detection, and ray-casting, which are all very important for developing a game. We also thought it was important for our project to be cross-platform during development and for the end users. Unity provides this initially, where other game engines do not. We wanted our game engine to be powerful and robust, and Unity fits this goal.

**Report**

Background.cs

This script is small, but is the essential initializer of the game. It sets the size of the map, and initializes the Map and Grid in its StartUp method. Inside Background's Update method, the user has the ability to left-click to place and remove walls, and right-click to add Agents to the map. This is done by translating mouse coordinates to world coordinates.

Agent.cs

The Agent script is used to create agents, and is the parent of the Player class. Agents have a heading, velocity, cell index, and radius. The Start method initializes these. Agent's Move method allows for Agent objects to move around the map by using the objects velocity, heading, and cell indexes of the grid. Only the Player object uses this method currently. The getLengthOfFeelers method returns an array of vectors that gives the current length of each feeler given the number of feelers needed, the max length of a feeler, and the view angle of the Agent object. Based off of the number of feelers and a view angle, the angle for each feeler is calculated to ensure the angular distance between each feeler is equivalent. After the direction of each feeler is calculated, we use Unity's Physics library to cast a ray in that direction starting from the radius of the Agent object, and going outward. These rays are limited to a length of the given max feeler length value. If the feeler is not colliding with anything, it will have a magnitude of the max feeler length, otherwise the magnitude will equal the distance from the collision to the Agent object. This script also contains two getter functions, for cell index and radius.

Player.cs

Player extends Agent. The Start method initializes the Player. Within the Update method, the game takes in keyboard input. This includes movement of the Player agent and debug keys. This method also gets the current length of the feelers, and current adjacent agents. The onGui method is called every frame to draw something onto the environment, and comes from the MonoBehaviours class. This is the method which draws all the debugging output. The first output rendered is the feelers. These are rendered by calculating their magnitude in camera space and drawing a box of this magnitude in the direction of the feeler. The next output rendered is the circle, which shows the radius of the adjacent agent sensor. The circle is just a texture scaled to the correct size. Pie slices are then calculated and drawn as another set of boxes, but of the size of the circle radius. Once these have been displayed, each agent in the nearest agents list is examined to see which pie slice it is located within and then labeled. To determine the pie slice, the angle is calculated by finding the dot product between the heading vector and the vector from the player to the agent. All of this debug information is then printed out to the left-hand side of the screen. Then follows the drawBox method, which takes in coordinates, box dimensions, an angle, and a pivot. The method then uses this information to draw a filled in black rectangle to screen, rotated around the pivot at the specified angle.

Grid.cs

Grid is an organized container for all of the agents on the map. It consists of a 2D array of cells, and each cell can contain multiple agents. Agents are added to a cell based off of their coordinates in the world space. The constructor generates a new grid of a given width and height. Then follow two inBounds methods, one taking in a vector of coordinates, and one taking in x and y coordinates. These methods check to see if the cell index of the given coordinates is within the bounds of the grid. The add methods will add a given agent to the grid. If the location given to these functions is not valid, they will return false. The remove methods will remove a given agent from the grid. They return false if a valid agent is not passed in. The move method is called when an agent moves from one grid cell to a new one. This updates the agent's location in the grid. The getNear method takes in an agent and a radius and returns a list of agents that exist within the given radius around the given agent. This method is used when trying to locate adjacent agents. Based on the given agent's center and radius, the method finds the range of grid cells that can be detected by the given agent. It then loops through each of these cells, and if other agents are found, they are added to the list of agents to be returned.

Map.cs

Map is similar to Grid, however it deals with the placement of walls instead of agents, and each cell can only contain one wall. The constructor generates the start map for each run of the program, which consists of a background and a border of walls around the edges. The createBorder method is the implementation of creating the border walls that is called in the constructor. The inBounds method verifies that a given wall's coordinates are in bounds of the map. The getCellIndex method returns the cell index the given wall is in. The cellIndexToWorld method translates a given cell index to a world coordinate. The addWall method verifies the coordinates of a given wall and then adds that wall to the map by calling placeWall. The removeWall method removes a selected wall from the map. The user can no longer see this wall, and the Player agent can now move where there once was a wall.

**Lessons Learned**

Overall, we learned that Unity is extremely helpful as it handles much of the functionality for game programming. Unity's interface is easy to use. It makes it simple to add textures and sprites, as well as create new scripts and attach them to objects. We also learned to fully read the directions before assuming we were completed with the assignment. We had most of the code written before we realized we needed to print out the debugging information to screen. Early on we learned that debug scripts only affect the editor within the engine, and not the actual game. Displaying GUI objects in Unity was also more difficult than we expected. We searched the internet and found tutorial videos in order to learn how to use these properly. During the development, we discovered several of Unity's useful functionalities, such as colliders.