Christopher Ross

Daniel Harrison

1) We picked XNA because it is a well documented engine that has an ample amount of tutorials. It lacks an integrated physics engine, but we could easily hook Farseer into it and start working on the project quickly. One of the members of the team also had some familiarity with the engine going in and this gave us a kick start. We had considered other engines, but felt that the structure that XNA provided would enable us to spend more time learning about AI and less time fighting with the framework.

The Project:

Playing the Game:

You are controlling a purple circle. WASD moves the player around the screen. left clicking with the mouse creates/destroys walls blocks and right clicking will create non-moving agents. Some moving agents will be spawned when the map is created. pressing P pauses and R resets your position to a random free spot.

The meat of the code:

This Code deals with the following areas:

* Gamplay.cs(What is in charge of "The Game")
* Map
* Sensors
* BadGuy.cs
* Player

For the sake of discussion we will talk about each piece individually.

Gamplay.cs:

The Gameplay.cs file is manly in charge of everything. There are a variety of different files that are in charge of getting you there, but everything that we would consider "the game" is deal with in that file. The first thing it does is setup different pieces of data; globals, the map, the player and the mobs. After that point the file is essentially a manager. It's update function will go call update functions on the game's different parts(map and mobs) as well as handling the game's input(either handling the input itself like when pausing or calling a function for the player to handle the input). The same thing is done when drawing. Gamplay.cs goes through each of the parts of the game and draws them, the order of the calls will dictate what shows up on top though, so special care has to be taken to ensure that the correct parts show up on top.

Map:

The map is created when gamplasy.cs starts. It is a procedurally generated map based on a grid. To start the map creates rooms(rectangular areas) on the map and random open spots. If this was it though, you couldn't move anywhere so we then continually clear out space until all open floor is attached. Finally we cover the border of the map with walls, unfortunately this may destroy some connections. The rooms are attached by selecting a point on the map that is in a room and a random direction and then moving in that direction while clearing out walls until it hits another clear spot. Before each step of the wall remover it will have a 10% chance of turning. Originally there was no heading and the wall remover always took a random direction, but this lead to less pathways and more cleared out spaces. The wall remover tended to eat out an area as opposed to moving to a new one. We also had the color of the wall pieces change based on it coordinates as to create different regions in the world. The map generation is good for now, but will be a point of interest in the future.

Sensors:

Each of the sensors is handled in a separate file. The feelers do a raycast from the center of the agent to an endpoint dictated by its angle and the angle of the agent. The RayCast takes in a function that will decide some of its behavior. This function must return 1 so that the raycast will continue to run and we can see what collision point is going to be the closest. Original it stopped at the first collision and used that distance, but we found that if the raycast would go through different objects, the first collision returned may or may not have been the closest one.

The area sensor was simply a static circle, where i reset its position to that of the players each step. It also was sensor type of physics object which meant that, it didn't run into objects the way normal physics objects would. An agent or wall could enter into it and a collision would be fired, but the circle wouldn't stop or impede the colliders progress. When collisions occur, the sensor maintains a list of body's that collided with the circle and then removes them from the list when they leave the area of the circle. From there it is simply performing the need calculations between the player's physics object and the list of physics object caught by the area sensor.

BadGuy.cs:

The mobs are pretty straightforward. They have a circle physics object and an instance of the different sensors. For the sake of this assignment the only sensor that effects the mobs behavior is the feelers. We created some basic ifs to direct the agents toward open areas. This is a good start, but will likely be part of future improvements.

Player:

The Player derives directly from the BadGuy class and as such the player gets all the same sensors. There two major distinctions between the player and the badguy, the player is hooked up to the HUD to display the sensor's data and the player override's the BadGuy's update function so that it takes in the keyboards input as opposed to determining its moves itself.

Conclusion:

Most of it was pretty straightforward, most of the work went into learning the engine and getting the calculations perfect. The engine worked they way we expected, it simply takes time to become familiar with it. We had some issues with passing around data though, in the future we would like to remove the Globals file if possible. There were a few issues getting the conversions right between XNA and Farseer, but this was quickly resolved once we became more familiar with the software. At first the Feelers seemed to work in a sporadic fashion, but it turns out we made some initial assumptions that turned out to not be true.