WLIAS TECHNICAL GUIDE

Project Details

Student 1 Name : Sean McCann
Student 1 ID : 16448004
Student 2 Name : Kacper Slowikowski
Student 2 ID : 16446386
Title : We Live In A Society
Finished on : 15/5/20
Supervisor : Alistair Sunderland

Introduction

This is a Simulation / Strategy game where four societies compete for resources on a tile based map. It runs on a self made graphics engine, using Lightweight Java Game Library (LWJGL) and OpenGL concepts. The map consists of four tile types. Each type gives a different bonus to the society that claims it. The map dimensions and amounts of each type of tile are predefined, however the shape and order of the tiles is randomised. The goal of the game is to either be the last society left on the map, or have the largest score at the end of the 50th turn. A player will take on the role of one of these societies and make choices for them. The enemy societies will be previously trained artificial Neural Networks created using the NEAT algorithm.

Motivation

Our motivation for this particular style of project came from a combination of interest of Simulations, graphical rendering, game development, and strategy games. This interest however would not grow into the project that we have made without following examples of projects with similar approaches.

Code Bullet is a Youtuber whose content revolves around creating games, and then training an AI to beat said games. He utilizes two main approaches: Q-Learning and the NEAT algorithm. We have adapted the NEAT algorithm approach in training of our own enemy AI.

Thin Matrix is a Youtuber whose content revolves around creating tutorials for various different programming libraries. From him we have learned about LWJGL and how to utilize it to create our own game.

These two combined sources with our interests developed into this project. A 3D isometric simulation/strategy game about societies competing for resources on a randomly generated map.

Research

The research aspect of this project fell within two primary areas. Firstly we needed to research the methodology that the societies would use in order to make their decisions. Initially we wanted to use an algorithmic approach borrowed from studies of interactive entities but it soon became clear that the scope of our game was greater than would be realistically feasible to carry on with our approach.

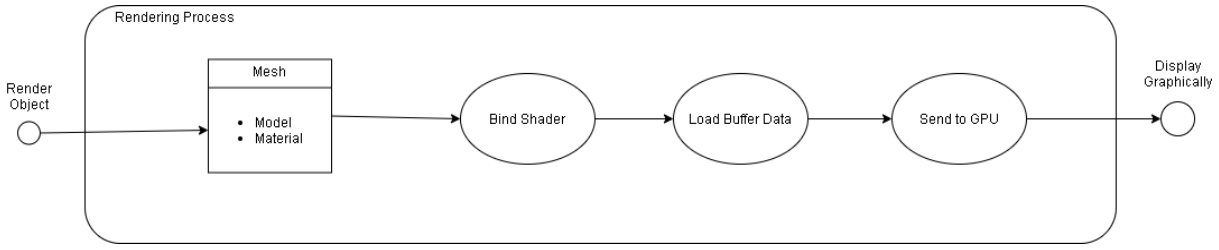
From there we felt that using some sort of machine learning approach would be the best option. We looked into a few different approaches such as Q-Learning in order to train our network but in the end we decided to use the NEAT algorithm as this is an approach which is growing in popularity especially within game environments. We conduct further research into implementing NEAT through youtube videos, and reading the official research paper.

Secondly, we had to research into how we would graphically represent our application. Initially we thought about using a simplistic 2D approach but in the end we decided that implementing a 3D engine would provide a better experience and would grant more options. We decided that we wanted to implement our project in Java so after looking at different options we settled on using the Lightweight Java Game Library (LWJGL) to develop our 3D engine. This library provides a Java API for OpenGL functionality.

Design

Rendering Process

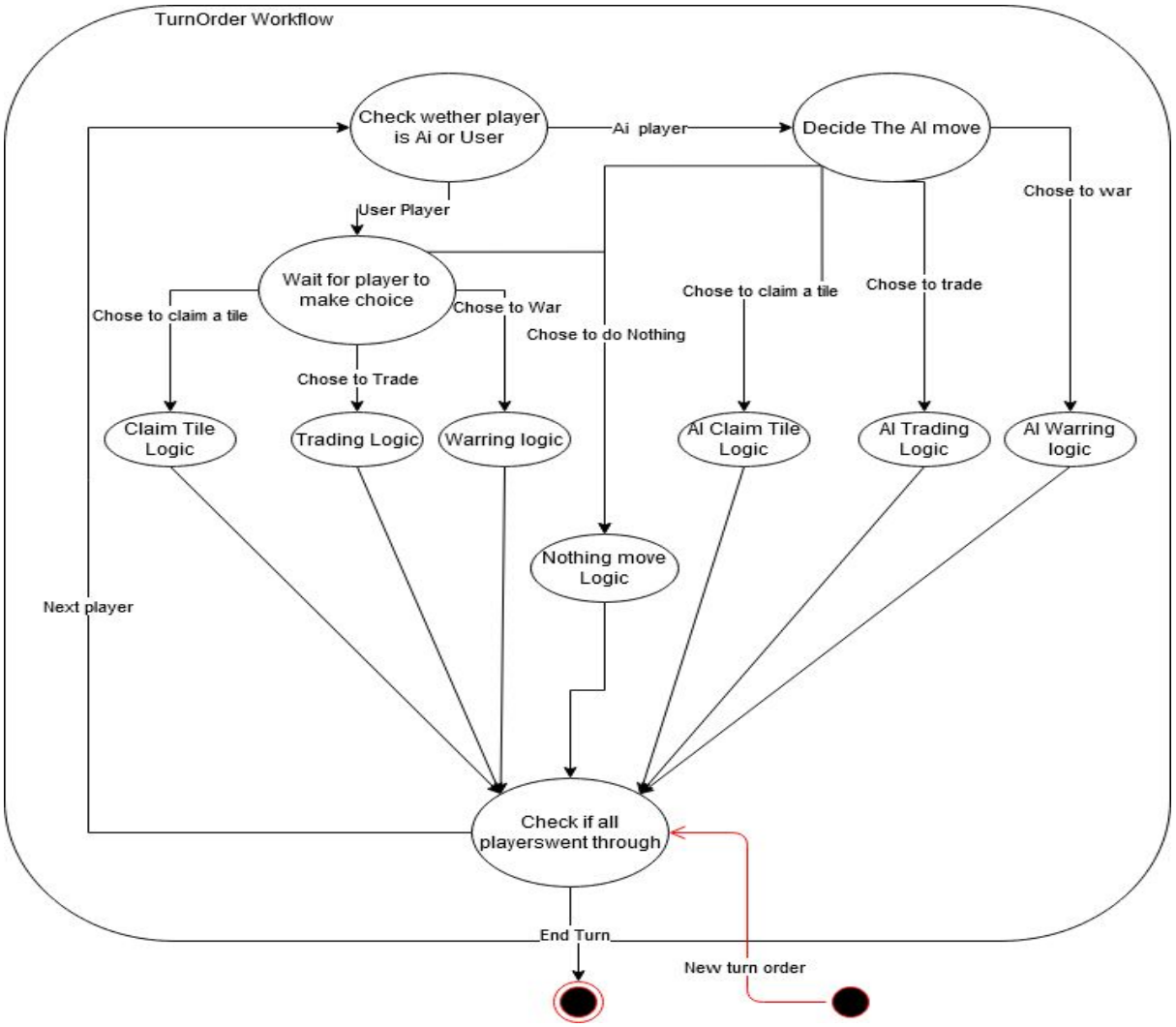
When rendering an object it follows this high level flow



Once an object is created we can make a request to render the object. An object contains a Mesh which contains both a Model and Material describing its shape and texture. When we call to render a shader script is loaded into memory and defined as active. Once the shader is initialised we can extract the buffer data from our object and pass that into our Renderer which loads the data as active and passes it to the GPU. The GPU reads this data as well as the shader script and performs calculations in order to display the object visually.

Game Logic Handler

The flow of our game is based on this simple design of a full turn within our game.



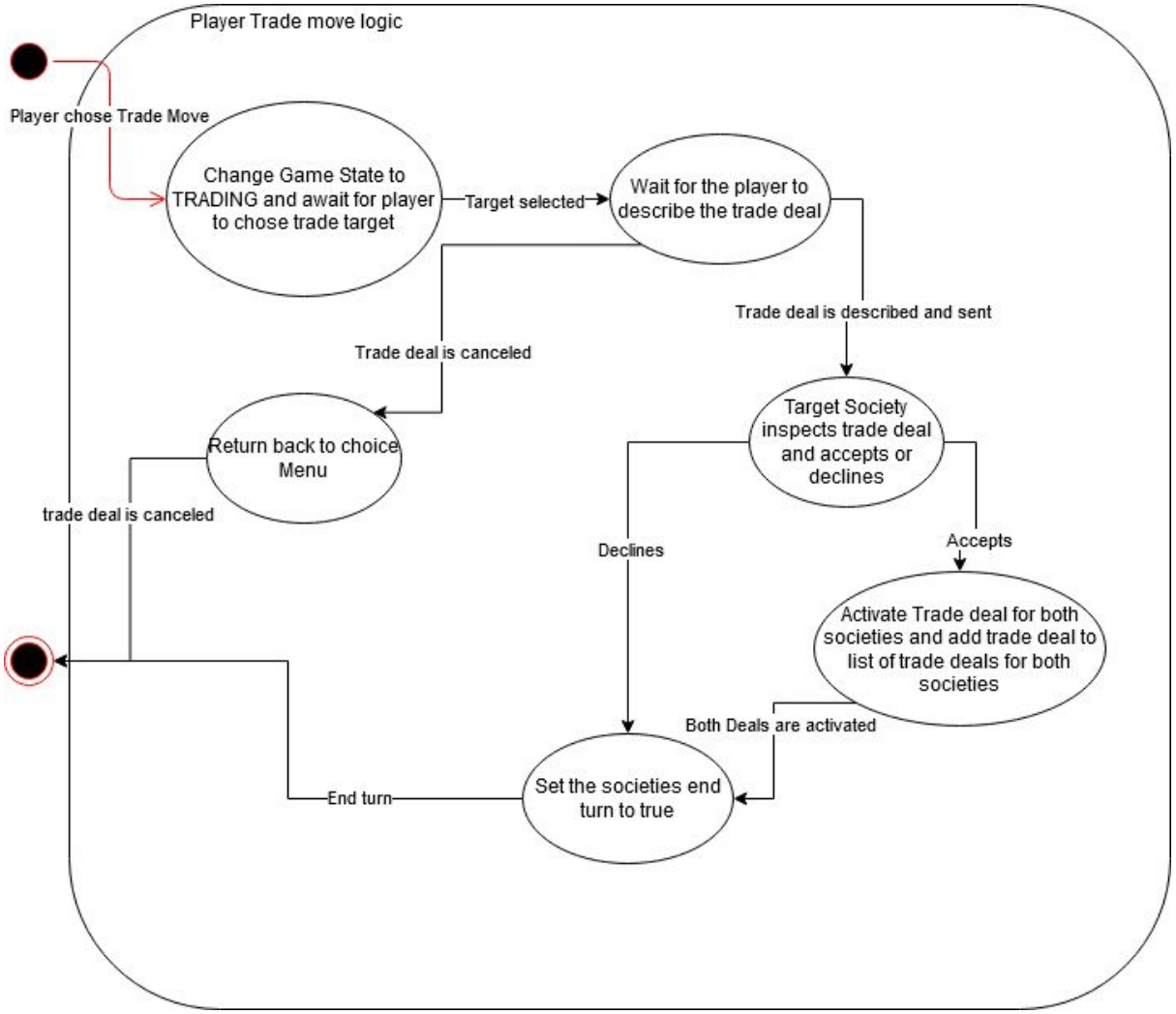
A turn order is decided by selecting societies which are still alive in the game and randomising it. The process then iterates through each society and checks whether or not that society is a user or an AI. Depending on which it is it goes into one of two Game States. PLAYER_CHOICE or AI_TURN.

In PLAYER_CHOICE the application will wait for the user to choose one of four moves. These moves will be displayed on screen to the user. It will then carry out that move and carry on through the iteration to either the next player or the end of turn.

In AI_TURN the application will, using the NEAT Artificial NN, decide which one of the four moves to carry out. It then also carries on to the next iteration in the turn order.

When all societies have chosen a move and the turn ends, the game checks the turn counter and the remaining societies left in play. If the game is not yet over a new turn order is selected and the above workflow begins again.

Example of User Move logic



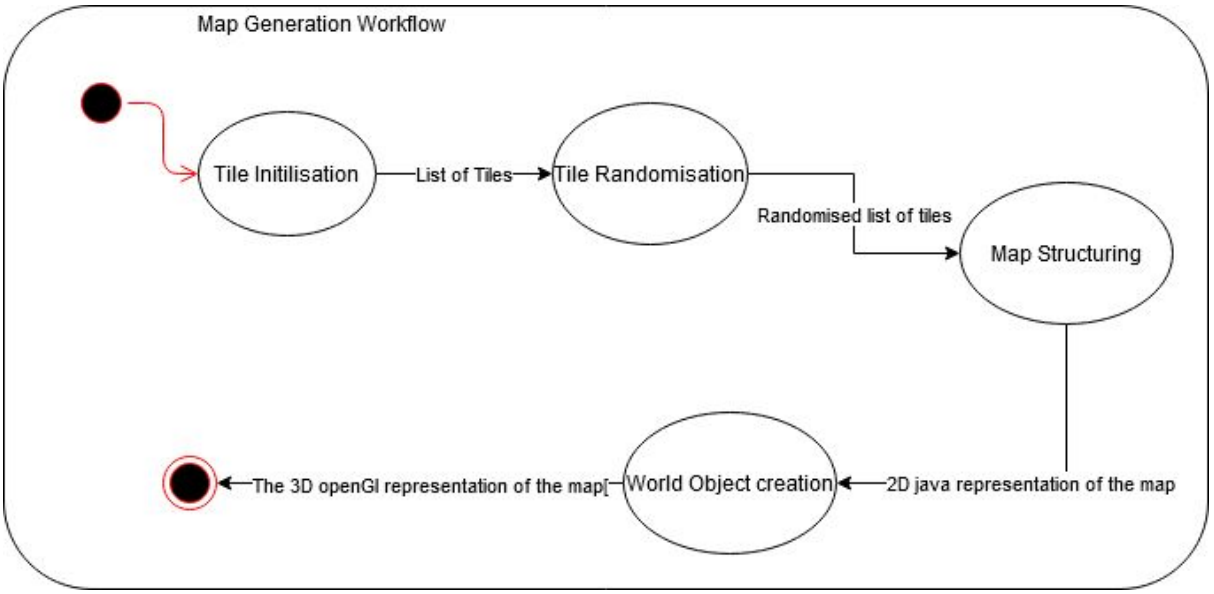
This is an example of the workflow that follows when a user decides to make a trade move. The game changes from the CHOICE_MENU state into TRADING state and awaits for the user input.

After a target society is selected for trading the trading menu is displayed on the screen. The user is then free to either cancel the trade deal and return back to the CHOICE_MENU state where they can choose a different move, or they describe the trade deal using the given menu and send it to target society for inspection.

The target society inspects the trade deal and decides whether or not it is beneficial to them. It then returns a response. If it is accepted both societies gain and lose the resources from the trade deal, and the end turn for the trade deal is set and it is added to a list of active trade deals each society has. The end turn flag is then set to true and the state is changed to GAME_MAIN.

Map generation Logic

The Map generation Design is also quite simple.



Based on previously defined land mass size and the amount of water, arid, fertile and plain tiles a list of tile objects are created. These objects then go through the process up above and get transformed into the map.

Implementation

Technologies used

- LWJGL
- Java
- Gradle

3D Graphics Engine

The first main component of the program is the 3D graphics engine. The process for rendering graphics is split into multiple stages. Firstly we have to incorporate our own miniature math package which handles custom Vector and Matrix objects and a large array of custom calculations that must be supported between each as these objects are the backbone of 3D rendering.

The main shape of objects are drawn using vertices while they are visually projected and manipulated using matrices. Object data is saved to meshes which hold a model defining the vertices of the objects and a material which holds colour and texture data relating to an object.

Once an object is defined we can create it to save its data into buffers. When we need to render we simply load our object into a custom made renderer which reads these saud buffers and loads them into a shader script. The shader script performs mathematical calculations of these buffer data values as a result the GPU will draw the object onto the screen.

When an object is no longer of use we can delete these buffers to free space. Shaders are written in GLSL format and come in two forms, vertex and fragment shaders. Upon program termination all remaining buffers are destroyed such as those relevant to the main menu and all active shaders are released.

Game Logic Handler

Map generation

At the start of a game a new random map is generated using the predefined parameters provided. Upon creation of the map each tile within it is given a Type and a number of food and raw materials resources. The amount given to the tile is decided based on what type of tile it is.

After the map is created and all tiles have been assigned their resources all societies are placed down on the map in random positions, however no society can start on a water tile. Each society begins with a population of ten people. Each person has a number of traits which influence their society's happiness and ability to war.

Turn

A full iteration through one game loop is called a turn. This means that after a turn all societies have made their moves. The turn order for the societies at play is randomised at the beginning of each new turn. The turn order can be seen in the top left of the screen at all times by the user. The turns are ordered in descending order. During a society's turn it is highlighted in yellow.

Every 2 turns the reproduction occurs. Reproduction is based on a genetic algorithm. This algorithm uses crossover and mutation to create new persons in a society with possibly new traits. This genetic algorithm also handles the ageing and death of persons within the society. The death ratios within the genetic algorithm are based of a study “*Risk of death by age and sex*” - Bandolier.org.uk.

After the reproduction step the program tries to identify whether the player is an AI or a User and head down two different workflows.

Player Turn

At the beginning of a user's turn the game enters a CHOICE_MENU state. While in this state a menu with 4 different moves will be shown on the screen. The program will then wait for the player to choose one of those moves.

When a move is chosen the program will change the game state to the appropriate state based on the move WARRING, CLAIM_TILE, TRADING, or simply skip to next turn if

“nothing” has been chosen. Each move within the game is accompanied by a hint which explains to the user what the steps to completing the move are. Once a move is completed the game loop iterates to the next player within the turn order.

AI Turn

At the beginning of an AI turn the game enters an aiTurn function. This function uses the neural network to identify which move to make based on the inputs provided to the Network. Each move is given an index from 0 - 3. Based on this index one of 4 game states is set AI_CLAIM, AI_WAR, AI_TRADE or AI_NOTHING. Once one of these is set the program goes into the workflows for each move and completes it.

Throughout the AI turn the player can see what the society is doing by watching the hints on screen.

Goal

The goal of the simulation / strategy game is for a society to either control all of the tiles provided or to have the biggest score at the end of the 50th turn. The score is determined based on a number of factors such as society's happiness, productivity, population, territory size.

Happiness is an integral part of the scoring function. It is a resource of a society which is not only based on the resources it has within its border but also based on decisions a society makes. If a society decides to go to war and loses, the happiness will decrease significantly.

NEAT Algorithm

Explanation of Neat

For the machine learning aspect of our project we implemented the NEAT algorithm. Neuroevolution of augmenting topologies (NEAT) is a dynamically evolving neural network implementation which is not confined to a static amount of hidden layers and nodes. The algorithm instead only begins with the required amount of inputs and output nodes and over time as it trains and evolves more nodes and connections will emerge.

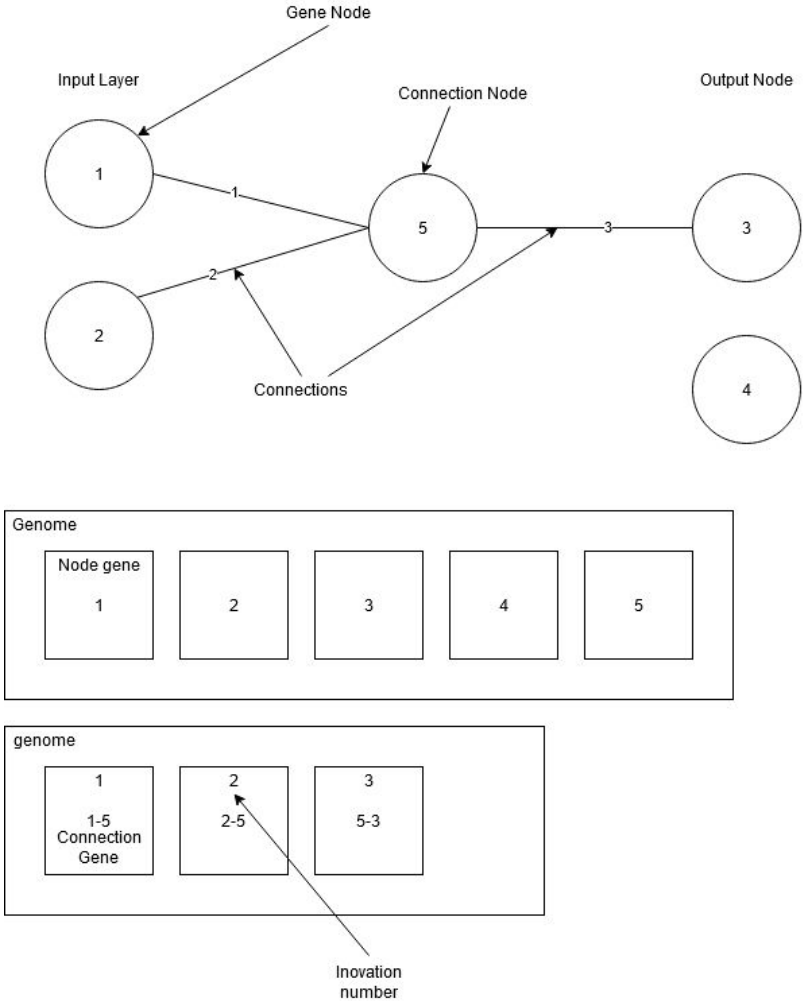
Implementation

We implemented this feature by running multiple game environments using Clients. A client is a structure which holds a unique neural network and saves its relevant score. Clients can be grouped together into species which are defined as networks of similar structure. The idea behind this is that over time certain structures may differ but both be high functioning. If we evolved structures of different species while also maintaining species that tend to score high we will come across better functioning networks quicker with a wide array of strategies.

When all of the clients have run a simulation and have obtained their score the topology will evolve. Clients will be sorted by their scores and a certain portion will be killed. Then species will select members to crossover and mutate similar to a genetic algorithm which in turn adds more members to the high scoring species. This is the main principle of NEAT which is to evolve organically in accordance to a genetic algorithm instead of fixed parameters When a

game is started the program will see if the NEAT structure is saved in the relevant resources directory. If the file is not present a new NEAT structure is created, else the file is loaded and this structure is used instead.

Client Structure



This is an example Structure of a client NEAT produces. The inputs genes consist of things that the client can see. An example of inputs in this project are, The territory size, army size, population size. Based on these inputs NEAT will try and identify what output should be triggered.

The outputs are the four moves a society can make in a turn. Claim Tile, Trade, War and Nothing. Each one of these moves either increases or decreases the overall score of the society.

When the Network is first initialised it does not have any connections or connection nodes. As the training progresses the network creates many of these clients and through mutation and cross over, it creates connections in those clients. A set of genes in a client is kept inside a genome structure. Connection genes are kept in a similar structure.

Each connection gene is given a unique Innovation number. This is done to improve the efficiency of the cross over function. When clients make connections between node genes and that same connection has been made already in a previous iteration. The new connection is simply replaced by the old connection with the old innovation number.

Training NEAT

We have opted for training our AI in two different environments. We have done this to ensure that the AI learns the basic rules of the game before we place into a competitive match. Those two environments are:

Single society training environment

This environment places only a single society onto the map. There the society learns about claiming tiles, different types of tiles and their bonuses, and it learns the goal of the game, i.e getting the highest score.

Multi-society training environment

This environment places four societies onto the map, much like if a user was playing the game. After multiple iterations of societies fighting against one another, they will start to discover warring and trading, and the situations in which they should war or trade.

In each training environment a game would be played. The score of the first society would be altered slightly based on whether or not they were the winner. Then using the NEAT genetic algorithm techniques such as cross over, and mutation new NN's would be created each with its own score. Then at the beginning of the next training session the best NN is chosen as the one being trained by society 1. The other societies get a random NN from the set of all NN's irregardless of their score.

Testing

Testing the engine

For testing the engine we used unit tests. However, these same tests could not be run on the CI/CD pipeline. An error when trying to initialise GLFW and graphical support framework, prevented the tests from being run upon the pushing of code. As such it was up to the developers to run tests locally before pushing to ensure that the graphical tests pass.

Unit Testing

Most of the project's testing was done by carrying out unit tests using JUnit 5. Most of these tests are tests to ensure that the logic for the non-graphical side of the project works as expected. The project is managed by Gradle so everything can be run automatically and handled easily,

User testing

The project also has a small number of user tests carried out. The users would play a few sessions of the application after being taught the core concept of the gameplay. Users that would then fill out a short survey in order to provide feedback on their impressions of the program.

CI/CD Pipeline

To ensure the quality and to streamline the project, a pipeline was set up using Git with three major steps.

Unit tests

This step ran all of the unit tests within the repository, upon a failed test the build would fail and we would be notified via Git.

SpotBugs tests

SpotBugs is a program which identifies bugs within Java code using code analysis. An example of which would be public variables which shouldn't be public or unused variables or just general bad coding practice. Similarly if this step was to return any errors the build would fail.

Checkstyle Tests

Checkstyle tests scan through the written code and ensure that the baseline syntactical rules of writing code were followed. These rules include, naming functions according to appropriate naming conventions, ending files with new line characters, or lines of code not extending past a certain amount of characters.

There are many different templates that can be loaded into the checkstyle checker. This project follows The Google Checkstyles template and maintains this style in order to make the code readable and maintain a consistent style.

Sample code

Game loop and checking for moves

Main game loop

```
private void gameLoop() {
 System.out.println("This is the Game Loop\n");
 while (!window.shouldClose()) {
   restarted = false;
   // Main game loop where each turn is being decided
   if (World.getActiveSocieties().size() > 0
       && state != GameState.TURN_END
       && state != GameState.GAME_PAUSE
       && state != GameState.GAME_OVER
       && state != GameState.GAME_WIN) {
     // Update the scores
     updateScores();
     // Check if the game is over
     checkGameOver();

     if (state != GameState.GAME_WIN && state != GameState.GAME_OVER) {
```

```
        // Reset the player choice
        ChoiceMenu.setChoiceMade(false);
        // Age everyone in each society
        if (Hud.getTurn() % AGE_FREQUENCY == 0) {
          for (Society society : World.getActiveSocieties()) {
            society.agePopulation();
          }
        }
        // generates a random turn order of all the societies in play
                                ArrayList<Society>    turnOrder    =    new
ArrayList<>(World.getActiveSocieties());
        Collections.shuffle(turnOrder);
        // update the Turn Order
        Hud.updateTurnTracker(turnOrder);
        // cycles through all societies in play
        for (Society society : turnOrder) {
           // closing of the inspection panel so that information is up to
date
          if (society.getSocietyId() == 0) {
            Hud.setSocietyPanelActive(false);
            Hud.setTerrainPanelActive(false);
          }
          World.setActiveSociety(society);
          // check and end Trade deals
          society.checkTradeDeal();
          // update happiness based on resources
          society.updateHappiness();
          // set the end turn flag to false
          society.setEndTurn(false);
          society.setMadeMove(false);
           // update and render the screen until the society finishes its
move
          // or the simulation closes
          while (!society.isEndTurn()
              && !window.shouldClose()
              && !World.getActiveSocieties().isEmpty()) {
            // Break this loop if the game is restarted form the game over
screen
            if (restarted) {
              break;
            }
            // Make AI Moves only if there is no player input
            if (training) {
              World.aiTurn(society);
            } else if (society.getSocietyId() != 0) {
              // There is Player Input and it is an AI society
              World.aiTurn(society);
            } else if (!ChoiceMenu.isChoiceMade()
                  // There is Player input and the Player has not made a
choice yet
                && state != GameState.GAME_PAUSE
                && state != GameState.GAME_OVER
                && state != GameState.GAME_WIN) {
              state = GameState.GAME_CHOICE;
            }
            update();
            render();
          }
          // Society reproduces and the notification loop is set
              // Need to check this, if we don't the state is reset to
reproducing and the main menu
          // is not rendered.
```

```
        if (state != GameState.MAIN_MENU
            && state != GameState.GAME_OVER
            && state != GameState.GAME_WIN
            && state != GameState.GAME_PAUSE
            && Hud.getTurn() % REPRODUCE_FREQUENCY == 0) {
          reproduceLoop(society);
          World.setActiveSociety(null);
        }
      }
    }

    // End the turn if the state is appropriate
    // If these states aren't accounted for there are game play bugs
    if (state != GameState.MAIN_MENU
        && state != GameState.GAME_OVER
        && state != GameState.GAME_WIN
        && state != GameState.GAME_PAUSE
        && !restarted) {
      state = GameState.TURN_END;
    }
  }
  update();
  render();
 }
}
```

Checking for move

```
private static void checkChoiceButtonClick(Window window) {
 // Check if left button is down
 if (Input.isButtonDown(GLFW.GLFW_MOUSE_BUTTON_LEFT) && Game.buttonLockFree())
{
   Game.resetButtonLock();
   // cycle through all buttons in list of choice buttons
   for (ButtonObject choiceButton : choiceButtons) {
     // check if the mouse is over the button
     if (choiceButton.isMouseOver(window) && choiceButton.isEnabled()) {
                                                                        if
(choiceButton.getLines().get(0).getText().getString().equals(CHOICE_BUTTON_NAM
ES[0])) {
         choiceMade = true;
         // War button was highlighted
         Game.setState(GameState.WARRING);
       } else if (choiceButton.getLines().get(0).getText().getString()
           .equals(CHOICE_BUTTON_NAMES[1])) {
         choiceMade = true;
         // Claim Tile button was highlighted
         Game.setState(GameState.CLAIM_TILE);
       } else if (choiceButton.getLines().get(0).getText().getString()
           .equals(CHOICE_BUTTON_NAMES[2])) {
         choiceMade = true;
         Hud.setTerrainPanelActive(false);
         Hud.setSocietyPanelActive(false);
         TradingMenu.setTradeDeal(new TradeDeal(0, 0, 0, 0));
         // Trade move button was highlighted
         Game.setState(GameState.TRADING);
       } else if (choiceButton.getLines().get(0).getText().getString()
           .equals(CHOICE_BUTTON_NAMES[3])) {
```

```
            choiceMade = true;
            // Nothing button was highlighted
            World.getActiveSocieties().get(0).setEndTurn(true);
            Game.setState(GameState.GAME_MAIN);
          }
          Hud.setSocietyPanelActive(false);
          Hud.setTerrainPanelActive(false);
        }

      }
    }
}
```

Game States

```
package game;

public enum GameState {
 MAIN_MENU,
 GAME_MAIN,
 GAME_PAUSE,
 GAME_CHOICE,
 WARRING,
 GAME_OVER,
 CLAIM_TILE,
 AI_CLAIM,
 TURN_END,
 AI_NOTHING,
 TRADING,
 REPRODUCING,
 DEALING,
 AI_WAR,
 GAME_WIN,
 AI_DEALING
}
```

Breakdown

The main game has an infinite loop which only exists when the player closes the game. At each iteration of the while loop a turn occurs. A turn starts by identifying the remaining players in the game and checking whether the game is over. If the game is not yet over the player choice is reset to nothing. If the turn has come for reproduction, the game state is changed to REPRODUCTION. Otherwise a player can make their turn as normal.

In the REPRODUCTION state a genetic algorithm is used to reproduce a society. A society consists of a number of persons. These persons all have a set of parameters which acts as genes. These genes include aggressiveness, productivity and such. Using crossover and mutation a new generation of people is born.

A User's turn is handled by the Checking for move function. Inside that function the program listens in to check whether one of four buttons displayed on screen has been clicked. If it has then the game will change states into either TRADING, DEALING, WARRING or

CLAIM_TILE. All of the game states are set up in the enumeration shown in the game states section.

Example Renderer class and appropriate Object

Gui Renderer

```
/**
* The type WorldRenderer.
*/
public class GuiRenderer {
 private static Vector2f DEFAULT_ROTATION = new Vector2f(0, 0);
 protected Shader shader;
 protected Window window;

 public GuiRenderer(Window window, Shader shader) {
   this.shader = shader;
   this.window = window;
 }


 /**
  * Render mesh.
  *
  * @param object the object.
  */
 public void renderObject(HudImage object) {
   initialise();
   bindObject(object);
   drawObject(object);
   unbindObject();
   terminate();
 }

 private void initialise() {
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
 }

 private void terminate() {
   glDisable(GL_BLEND);
 }

 private void bindObject(HudImage object) {
   // Bind Mesh VAO
   GL30.glBindVertexArray(object.getMesh().getModel().getVao());
   // Enable Index 0 for Shaders (Position)
   GL30.glEnableVertexAttribArray(0);
   // Enable Index 1 for Shaders (Colour)
   GL30.glEnableVertexAttribArray(1);
   // Enable Index 2 for Shaders (Texture)
   GL30.glEnableVertexAttribArray(2);
   // Bind Indices
                         GL15.glBindBuffer(GL15.GL_ELEMENT_ARRAY_BUFFER,
object.getMesh().getModel().getIbo());
   // Set Active Texture
   GL13.glActiveTexture(GL13.GL_TEXTURE0);
   // Bind the Texture
   GL13.glBindTexture(GL11.GL_TEXTURE_2D,
```

```
        object.getMesh().getMaterial().getImage().getTextureID());
    //Bind Shader
    shader.bind();
    // Set Uniforms
    setUniforms(object);

  }

  private void setUniforms(HudImage object) {
    setModelUniform(object);
    setProjectionUniform();
    setColourOffsetUniform(object);
  }

  private void setColourOffsetUniform(HudImage object) {
                                          shader.setUniform("colourOffset",
object.getMesh().getMaterial().getColorOffsetRgba());
  }

  private void setModelUniform(HudImage object) {
    shader.setUniform(
        "model",

        Matrix4f.transform(
                                  new   Vector3f(object.getPosition().getX(),
object.getPosition().getY(), 0f),
            object.getRotation(),
            new Vector3f(object.getScale(), 1)));
  }

  protected void setProjectionUniform() {
    shader.setUniform("projection", window.getOrthographicMatrix());
  }

  private void unbindObject() {
    //Unbind Shader
    shader.unbind();
    // Unbind Indices
    GL15.glBindBuffer(GL15.GL_ELEMENT_ARRAY_BUFFER, 0);
    GL30.glDisableVertexAttribArray(0);
    GL30.glDisableVertexAttribArray(1);
    GL30.glDisableVertexAttribArray(2);
    // Unbind VAO
    GL30.glBindVertexArray(0);
  }

  private void drawObject(HudImage object) {
    GL11.glDrawElements(
        GL11.GL_TRIANGLE_STRIP,
        object.getMesh().getModel().getIndices().length,
        GL11.GL_UNSIGNED_INT,
        0);
  }}
```

Shaders

Vertex Shader

```
#version 330 core

in vec2 position;
in vec3 color;
in vec2 textureCoords;

out vec3 passColour;
out vec2 passTextureCoords;

uniform mat4 model;
uniform mat4 projection;

void main() {
   gl_Position =  projection * model * vec4(position, 0, 1.0);
   passColour = color;
   passTextureCoords = textureCoords;
}
```

Fragment Shader

```
#version 330 core

in vec3 passColor;
in vec2 passTextureCoords;

out vec4 outColor;

uniform sampler2D tex;
uniform vec4 colourOffset;

void main() {
   outColor = texture(tex, passTextureCoords) * colourOffset;
}
```

Example Object

```
public class ButtonObject extends HudObject {
 private static final Vector4f DEFAULT_INACTIVE_COLOR_OFFSET = new
Vector4f(1, 1, 1, 1);
 private static final Vector4f DEFAULT_ACTIVE_COLOR_OFFSET = new
Vector4f(0.6f, 0.6f, 0.6f, 1);
 private Vector4f inactiveColourOffset =
DEFAULT_INACTIVE_COLOR_OFFSET.copy();
 private Vector4f activeColourOffset = DEFAULT_ACTIVE_COLOR_OFFSET.copy();
 private Vector4f disabledColourOffset = new Vector4f(1, 1, 1, 0.2f);
 private boolean enabled = true;

 /**
  * Instantiates a new Button object.
  *
  * @param backgroundMesh the background mesh
  * @param text           the text
  * @param edgeX          the edge x
  * @param offsetX        the offset x
  * @param edgeY          the edge y
```

```
   * @param offsetY        the offset y
   */
  public ButtonObject(RectangleMesh backgroundMesh, Text text, float edgeX,
float offsetX,
                      float edgeY, float offsetY) {
    super(backgroundMesh, text, edgeX, offsetX, edgeY, offsetY);

this.getHudImage().getMesh().getMaterial().setColorOffset(inactiveColourOf
fset);
  }

  public ButtonObject(RectangleMesh backgroundMesh, float edgeX, float
offsetX, float edgeY,
                      float offsetY) {
    this(backgroundMesh, new Text(""), edgeX, offsetX, edgeY, offsetY);
  }

  public void update(Window window) {
    updateColourOffset(window);
  }

  private void updateColourOffset(Window window) {
    if (!enabled) {

this.getHudImage().getMesh().getMaterial().setColorOffset(disabledColourOf
fset);
    } else {
      if (isMouseOver(window)) {

this.getHudImage().getMesh().getMaterial().setColorOffset(activeColourOffs
et);
      } else {

this.getHudImage().getMesh().getMaterial().setColorOffset(inactiveColourOf
fset);
      }
    }
  }

  public boolean isMouseOver(Window window) {
    return this.getHudImage().isMouseOver(window);
  }

  /**
   * Destroy meshes.
   */
  public void destroy() {
    this.getHudImage().destroy();
    for (HudText line : getLines()) {
      line.destroy();
    }
  }

  /**
   * Enable the button.
   */
  public void enable() {
    this.enabled = true;
    for (HudText line : this.getLines()) {
      line.getText().setAlpha(1);
    }
  }
```

```
/**
 * Disable the button.
 */
public void disable() {
  this.enabled = false;
  for (HudText line : this.getLines()) {
    line.getText().setAlpha(disabledColourOffset.getW());
  }

}}
```

Breakdown

This example piece of code shows the structure of a custom object for a button. Buttons can be given a background mesh to define their size, texture and text. Offsets must also be given to the object as it is written as a 2D ui element which means that when the window is resized we need to use the offsets in order to maintain its position and size on screen regardless of the aspect ratio. Buttons can either be enabled or disabled which dictates whether they are clickable.

When a button is disabled by default they are given a lesser alpha value which makes them transparent. During their update function a calculation is made to check whether the mouse is hovering over the button and in turn their overlay colour will change to notify the user. Since this object contains a mesh during the rendering stage its data will be passed to the renderers and passed to the GPU alongside a shader script in order to display the object on screen.

We can see in the renderer class it has an assigned shader and window to know which window and shader script to use when rendering. During the *renderObject* method we initialize certina openGl parameters to enable alpha blending. We then bind the object which reads its data from its VBOs and passes then to the shader script. In the shader scripts we pass in the position of the object, the projection matrix of the window, the model view matrix for the object, its colour offset and texture coordinates,. Finally within the fragment shader it receives this data and decides how to display the object to the screen.

Map Generation

```
/**
* The type Map generator.
*/
public class MapGeneration {

 private static final int HORIZONTAL_WATER_PADDING = 2;
 private static final int VERTICAL_WATER_PADDING = 2;
 /**
  * The Land mass size x.
  */
 private static final int DEFAULT_LANDMASS_SIZE_X = 10;
 /**
  * The Land mass size y.
```

```
 */
private static final int DEFAULT_LANDMASS_SIZE_Y = 10;
/**
 * The Number of land masses.
 */
private static final int DEFAULT_AMOUNT_ARID_TILES = 30;
private static final int DEFAULT_AMOUNT_FERTILE_TILES = 30;
private static final int DEFAULT_AMOUNT_WATER_TILES = 15;
private static final int DEFAULT_AMOUNT_PLAIN_TILES = 25;
/**
 * The Map of ordered tiles.
 */
static Tile[][] simulationMap;
/**
 * The List of tiles.
 */
private static ArrayList<Tile> tiles = new ArrayList<>();
/**
 * The Map size x.
 */
private static int mapSizeX;
/**
 * The Map size y.
 */
private static int mapSizeY;
/**
 * The Land mass maps.
 */
private static Tile[][] landMassMap;
private static int landMassSizeX = DEFAULT_LANDMASS_SIZE_X;
private static int landMassSizeY = DEFAULT_LANDMASS_SIZE_Y;
private static int amountOfAridTiles = DEFAULT_AMOUNT_ARID_TILES;
private static int amountOfFertileTiles = DEFAULT_AMOUNT_FERTILE_TILES;
private static int amountOfWaterTiles = DEFAULT_AMOUNT_WATER_TILES;
private static int amountOfPlainTiles = DEFAULT_AMOUNT_PLAIN_TILES;


/**
 * Create map.
 * Generates landMass tiles in accordance to the tiles provided.
 * Shuffles them to create random terrain.
 * sets the simulationMap to the joining of the landMasses.
 */
public static void createMap() {
  generateTiles();
  Collections.shuffle(tiles);
  landMassMap = generateLandMass();
  setSimulationMap();
}

private static void generateTiles() {
  tiles.clear();
  // stores the amount of tiles that are suppose to be in a land mass
  int reservedNumberOfTiles = landMassSizeX * landMassSizeY;
  // stores the amount of tiles that are provided to us.
  int actualNumberOfTiles = amountOfAridTiles + amountOfFertileTiles
      + amountOfPlainTiles + amountOfWaterTiles;
  // Check if the amounts are equal
  if ((reservedNumberOfTiles != actualNumberOfTiles)) {
     throw new AssertionError("Number of reserved Tiles does not equal
the Actual "
                + "number of tiles provided.\nActual Tiles = " +
```

```
actualNumberOfTiles + "\n"
        + "Reserved Tiles = " + reservedNumberOfTiles);
   }
   // initialise the tiles and add them to a single array
   for (int i = 0; i < amountOfAridTiles; i++) {
     Tile tile = new AridTile();
     tiles.add(tile);
   }

   for (int i = 0; i < amountOfFertileTiles; i++) {
     Tile tile = new FertileTile();
     tiles.add(tile);
   }

   for (int i = 0; i < amountOfPlainTiles; i++) {
     Tile tile = new PlainTile();
     tiles.add(tile);
   }

   for (int i = 0; i < amountOfWaterTiles; i++) {
     Tile tile = new WaterTile();
     tiles.add(tile);
   }

 }

 /**
  * Sets the mapSizeX.
  * Sets the mapSizeY.
  */
 public static void setSimulationMap() {
    /* This function adds all the previously made terrain onto a single
canvas
   it does so by adding the terrain one by one to a newly constructed 2d
array
    each landmass has the same dimensions so we know that each land mass
will be
     a distance equal to the previous landmass length away from the
leftmost edge of the 2d array
    to create some padding we also add a layer of water between each
landmass */

   // Sets the overall size of the Canvas(Map)
   mapSizeX = landMassSizeX + HORIZONTAL_WATER_PADDING;
   mapSizeY = landMassSizeY + VERTICAL_WATER_PADDING;
   simulationMap = new Tile[mapSizeY][mapSizeX];

    /* Since its a 2d array we will consider each row to be y co-ord and
each column to be x
   This means the top right corner is the point (y0,x0)
    y comes first as when accessing the array we will be accessing the row
first
   and column second. */

   setUpPadding();
   setUpLandMass();

 }

 private static void setUpPadding() {
   // this for loop creates the outer ridge of water
   for (int y = 0; y < mapSizeY; y++) {
```

```
      for (int x = 0; x < mapSizeX; x++) {
        if (x == 0 | x == mapSizeX - 1 | y == 0 | y == mapSizeY - 1) {
          Tile edgeTile = new WaterTile();
          simulationMap[y][x] = edgeTile;
        }
      }
    }
  }

  private static void setUpLandMass() {
    // y position of the landmass
    int landMassYPos = 0;
    for (int y = 1; y < landMassSizeY + 1; y++) {
      // x position of the landmass
      int landMassXPos = 0;
      for (int x = HORIZONTAL_WATER_PADDING - 1;
           x < landMassSizeY + HORIZONTAL_WATER_PADDING - 1; x++) {
        /* places a tile from a landMass in the appropriate y,x co-ords
        */
        simulationMap[y][x] = landMassMap[landMassYPos][landMassXPos];
        landMassXPos++;
      }
      landMassYPos++;
    }


  }

  /* Generates a list of LandMasses
   LandMass is a 2d array of tiles which represent the different land
masses */
  private static Tile[][] generateLandMass() {
    int tileCounter = 0;
    Tile[][] currentLandMass = new Tile[landMassSizeX][landMassSizeY];
    for (int i = 0; i < landMassSizeY; i++) {
      for (int j = 0; j < landMassSizeX; j++) {
        currentLandMass[i][j] = tiles.get(tileCounter);
        tileCounter++;
      }
    }
    return currentLandMass;

  }

  /**
   * Get map of ordered tiles tile [ ] [ ].
   *
   * @return The 2d Array representation of the map. Tile[][].
   */
  public static Tile[][] getSimulationMap() {
    return simulationMap.clone();
  }
}
```

Breakdown

This is the code that generates the tile array, simulation map, which is later used when creating the World Map used inside the game. It uses the predefined parameters for the land mass size and the amounts for the different types of tiles to generate the simulation map. These numbers can be altered but must always follow the rule of

*The combined number of arid, fertile, plain and water tiles must equal land mass sizeX * land mass size*

If this rule is not met the program will not start. If the amounts are correct the program initialises the appropriate amount of each tile and adds it to an array. This array is then randomised to create a completely random landmass.

The program then creates the 2D Array with a border of water surrounding the landmass. This was done to give the landmass a more realistic feel. Once the representation is finished it is ready to be called by the World class to be referenced when creating the world map which reads in the list of tiles and converts them into 3D tile objects for LWJGL to render..

Problems encountered / resolved

During the development of this application we had encountered a number of different issues, some of which we managed to resolve and others which still remain within the program. Firstly we encountered an issue when training the NEAT where repeatedly creating and destroying objects using our engine would cause a memory leak and eventually crash the program. We were unable to resolve this issue so unfortunately when training we had to resolve to merely displaying a notification screen while the game logic executes in the background.

The next issue we have that still exists within the application is that creating new Vertex Array Objects (VAO) which hold buffer data for objects is quite slow. Because of this rendering text and creating the world causes a noticeable lag to the application. We tried different ways of fixing these issues such as updating the buffers instead of deleting and creating new ones but nothing seemed to resolve the issue. When setting up the Git CI/CD pipeline we encountered an issue where openGl functionality could not run on the gut machines and would fail every time. To resolve this issue we enabled a checker to see whether the machine running the tests supports video capabilities, if so we can test graphical tests otherwise we only test logical tests.

Finally the main issue we encountered was figuring out ways to optimise the speed of the application while in game. One simple feature we implemented was mipmapping and anisotropic filtering. These features reduce the quality of rendered images based on the distance the camera is from the image, this saves on rendering time as less detail needs to be rendered every frame. The biggest optimisation we incorporated though was frustum culling. This is a graphical method which calculates a 3D frustum around the camera based on its position and rotation. Any objects which are deemed to lay inside of the frustum will be rendered while everything outside of the frustum will not. This was a massive improvement as no matter how large the map was a stable frame rate could be maintained.

Final product

Opinion of Final Product

The final product almost fully complements what the project set out to achieve. It is a smooth running simulation application which captures the developers interest in Graphical rendering technologies, game development and neural network technologies. It allows the users to directly interact and face against AI which are also trained within the same application.

What could have been done differently

If we were to start this again we would have focused more on the core game design before looking into graphical options. What we did was have a general idea and begin coding the environment before knowing exactly if we could achieve what we initially proposed. Over time it became clear that creating the 3D engine would be a far larger part than we initially intended and thus took away from what we could realistically achieve. We settled on implementing more general and simplistic gameplay features in order to achieve a complete end to end experience that would largely fulfill the initial proposition. Perhaps if we managed our planning better initially we would have been more efficient in the development process and could have added more features that would have heightened the user experience and interactivity.

Future work

There are a few features we would wish to improve on and implement in the future. While some of these features were initially proposed and expected to be added due to time constraints they were unable to be implemented. Firstly there are a number of visual improvements and graphical additions we would wish to see. These would be things such as adding light sources with shading and adding the ability to import and render 3D models such as models for resources on the map. We feel that this addition would greatly increase the interactivity and understandability.

Secondly we feel like our current usage of the NEAT structure is too restrictive, due to time constraints we limited the amount of inputs and outputs or program processes in order to save training time. In the future we would like to supply more inputs and outputs which would make our societies more flexible. We would like to add a database that would save statistics of games. Using a database we could analyze statistics and better show how our algorithm would evolve over time.

Lastly we would also like to create the map using a more sophisticated and realistic method such as Perlin Noise. This would provide a wider array of more diverse maps and terrain patterns. Lastly we would like to have added more openAL functionality. OpenAL is the audio handling library we incorporated into the project but currently it only handles background music, we think adding sound effects for interaction would greatly increase visceral engagement with the user.