Abstract and Game Overview

The purpose of this project was to produce a 3D game that depends on graph theory, procedural world generation, and artificial intelligence. The game was also intended to provide a good playing experience, as measured by beta testers.

The game is a maze-based world that the player has to explore in first person, all the while contending with the demands and map alterations of an unfriendly, powerful being (the game AI) in control of the situation. Communication with the AI occurs via a text interface (allowing me to avoid the need for voice actors).

The player starts the game in an unfamiliar grey room with 1-4 doors leading out to hallways. The hallways lead to identical rooms – these hallways and doors constitute the maze, and will be designed with a scientific simplicity and blandness in mind. As the game progresses, the player will have to decide whether or not to comply with the increasingly strange demands issued by a being that seems to be in control of this place. If the player complies, the game will become noticeably easier, with the being providing helpful hints and shortcuts. If the player does not comply, the being will become increasingly hostile, thwarting the player at every turn: locking the player in a room for a few minutes, spinning the maze to confuse orientation, moving hallways in the maze if the player is on the right path, and lengthening hallways.

The game should leave the player with the sense that they must choose between their own agency and winning the game – they can comply in order to win, with the downside that the being will mock them as weak for doing so (and the ending will be less satisfying).

Completed Deliverables (with technical details)

The game produces a convincing, random maze on each playthrough and the maze generation is high performance.

1. The base abstraction for the maze is a graph. More specifically, the maze is represented as an N by M matrix of graph nodes where the nodes can only be connected to their direct neighbors in the matrix (this is intuitively like a maze, where a room can only be connected to adjacent rooms).
2. I wrote my own barebones, efficient graph library in C# (contained in Graph.cs in the source code) to represent this abstraction. The Graph class supports basic graph operations such as adding and removing edges and nodes and finding paths between two nodes. I added a GridGraph type that contains the data structures needed to represent the graph described in (1). The grid starts out with each node connected to all of its neighbors.
3. The graph library includes an implementation of randomized Kruskal’s algorithm to produce a random spanning tree over the original GridGraph.
4. At the beginning of each game, the graph library is used to generate a new GridGraph and a random spanning tree. Then the main application iterates over the graph and produces a new room at each node and a new hallway at each edge.
5. This requires calculating the world positions of each room and hallway, which can be done straightforwardly from the coordinates of the graph node in the grid (with a single
parameter determining the distance between rooms). This approach was inspired by CatlikeCoding’s Maze Unity tutorial.

The controller AI communicates with the player correctly.

1. To handle player-AI communication, I created abstractions for a communication channel and a player response (see the AI/CommuncationChannels folder and the AI/PlayerResponse folder). The AI has access to a set of functions that it can use to open a channel, check for a response, get the response, and end the communication. A handler function is called after communication is complete to check the response and decide how to react.

2. There are two types of responses: movement and text. The AI can either receive a string response or a path that the player walked while the communication channel was active. The logic of checking for the right response is contained in the channel. The reactions for negative/positive outcomes are contained in AiPlayerInterchange objects.

3. This is the basic anatomy of an AI request to the player:
   a. The AI activates a given channel and sends a string as a message on the channel.
   b. The AI sets up an AiPlayerInterchange to contain responses and the expected response from the player.
   c. The AI queries the channel at every frame and does nothing until a response is received (to avoid overlapping requests).
   d. Once it is received, the response handler is called, the response is checked, and the interchange object is used to send back the correct string.

The controller AI changes the map in a variety of ways without breaking it.

1. Shortcuts and “longcuts”. The AI has methods to add a shortcut or a “longcut” to the maze given the player’s coordinates. The AI will check if any adjacent room provide a shorter path to the exit (using pathfinding methods of the underlying graph). If one does, it will add a hallway to that room and remove the hallway that originally led to the exit (reverse the logic for “longcuts” – it will try to find a longer path that does not go through the current room).

2. Signpost hints. The AI can use a pathfinding method from the graph to determine which edge exiting the player’s current node leads to the exit node. A signpost is constructed with the correct rotation and spawned below the floor – it then rises via a simple animation. The signpost always rises from the quadrant of the room that the player is not in.

3. Coordinate text on walls. 3d text can be added to the walls giving the coordinates of the cell. The built-in 3d text object in Unity is visible through other objects and from behind, so I had to use an open source shader for Unity that fixes this problem (Resources/materials/3DTextOneSided.shader).

4. Lengthening hallways. The distance between rooms in the maze is controlled by a single floating point public parameter. In order to lengthen the hallways, the new length is chosen, and on every frame the code linearly interpolates between the new and old length to calculate the correct positions of all the objects in the maze. This creates the illusion of a smooth hallway lengthening from the player’s perspective.

5. Shaking and spinning the maze. A separate ObjectMover class was created for basic animations on unity objects (so as to decouple the animations from the objects.
themselves). These animations use a similar approach to (2), linearly interpolating towards the final position (or final rotation) of the object, creating the illusion of smooth movement. These animations are then applied to the maze or the player.

6. **Transforming maze into final states for endings.**
   a. Standard ending: The AI uses an ObjectMover to raise the player above the maze and starts randomly destroying/regenerating the maze to show the player this process.
   b. Friendly ending: The AI regenerates the maze as a straight line where the doors close behind the player.
   c. Hostile ending: The AI regenerates the maze as a 3 by 3 grid where the outer nodes are connected in a ring (a circular maze).

The controller AI responds correctly based on player’s action, and the AI generates a wide range of randomized requests for the player.

1. The AI’s actions at any given time are determined by the state it is in (stored as a simple enum value). These states are VeryFriendly, Friendly, Neutral, Hostile, and VeryHostile. It can perform two types of actions: A request, which gets some sort of feedback from the player, or a reaction, which expects no feedback.
2. After receiving a response from the player, the AI checks to see whether the response was correct (as described above). The “number of infractions” is stored as an integer that loses value with each incorrect response and gains value with each correct response. At certain thresholds, the AI will change state (e.g. at -2 it will change from Neutral to Hostile).
3. Upon the AI’s initialization, it creates lists of actions that it can take in each state (these are represented by lists of C# delegate functions). Using C#’s built-in reflection functionality, the AI object iterates through its own method names and adds methods to a state’s list if the beginning of the method name matches the state name (for example, the method “Neutral_Request_Action” would get added to the Neutral state’s request action list).
4. The AI will also add text-only requests and reactions by parsing the text files in the Resources/GameLines directory (in the requests/ or reactions/ subdirectory) into interchanges. The AI will parse all the files in the subdirectory with a given state’s name and add an action for all of those interchanges (this is done using functions in the GameLinesTextGetter utility).
5. Every time the player enters a room, the AI will randomly select a reaction or request from the current state’s action list and execute it, creating the appearance that it is changing behavior based on the player’s actions.

The game produces the desired reaction in players.

1. Of 10 beta testers who tried out the game, about 8 reported a positive experience. They did provide a variety of suggestions for changes, which are discussed in the next section.

**Tester feedback and suggested changes**

These were the major pieces of actionable feedback I received from people who tested the game:
1. There was a bug on Mac OS X that did not appear in my testing of the game on Windows 10. Due to a null reference, the AI would sometimes lock the player in a room permanently, requiring a full restart.
2. Some players wanted more intermediate goals to keep the game engaging and interesting.
3. The AI could be cheated by hugging the walls, causing it to never detect the player entering a room.
4. Some suggested changing the breadcrumbs that can be dropped by the player into something more interesting or realistic looking.
5. The game could only be restarted by closing and reopening the app. Players wanted a simpler way to restart.
6. The controls were slightly too sensitive. Players wanted less sensitive controls.

I was able to fix (1), (3), (5), and (6) by changing some parameters within the code and tracking down the source of the bug on Mac (it had to do with the way newlines in text files were handled differently). Suggestions (2) and (4) could be areas for future work (see that section below).

Additional notes on the code and implementation

Technology used
The entire game was created using the Unity editor and C# code. Version control was handled by github. See the full source code at [https://github.com/iangonzalez/mouse-in-a-maze-game](https://github.com/iangonzalez/mouse-in-a-maze-game)
See the README for more information on where to locate various parts of the project.

Parts of implementation not mentioned in deliverables
There were a variety of aspects of the game that were not covered by the descriptions of the deliverables. Below are short summaries of some of the other notable portions of the game that had to be implemented:
   • Most of the objects in the game had to be created using the default Unity objects available in the editor. Some of the objects were taken from free asset packs and modified to work within my game.
   • The player’s functionality was fairly complex. The player object has to move according to user input, keep track of where it is in the maze (which requires an OnCollision function), freeze when in communication with the AI, and determine where (and if it is possible) to drop breadcrumbs.
   • Many aspects of the game’s appearance required precise tweaking, such as picking fonts for the AI, placing lighting and choosing lighting types, and choosing textures for various objects.
   • The game required high-level management logic to start the initial game state correctly and determine when to end the game. This code also had to fade the screen to black at the correct times.

Licensing
Some of the assets used in this project (certain textures and objects) were downloaded for free from the asset store. However, although they were free, the Unity Asset Store license agreement does not allow for public redistribution of those assets. If you would like to use the
assets in Assets/Building Assets or Assets/Free SteelLadder Pack, please download them from the Unity Asset store so as to credit the original creators. The two asset packs are Modular Abandoned Slaughterhouse and Free Steel Ladder Pack. These assets are not hosted on my github either.

Other than that, this project is entirely open source, and any part of it can be copied and redistributed at will.

**Future work**

This game is far from complete, and I plan on continuing to improve it after turning it in for this senior project. The first place to start will be the feedback that I was not able to get to due to the complexity of adding these features.

The game could definitely use more intermediate checkpoints, as mentioned in the tester feedback above. An important part of keeping a player engaged in any game is providing some sort of visible measure of success that gradually increases over time. An interesting way to do this for this game would be to add checkpoints in the maze. These could be implemented as arbitrary rooms in which the AI congratulates the player on making it this far and blocks the way back, thus letting the player know they’re getting closer. Another option would be to have multiple floors of the maze, and each “checkpoint” would be reaching the ladder to the next level.

Another area of the game that needs serious work is the appearance. I tried my best given my lack of artistic talent, but I think the best way forward is to work with a designer to enhance the aesthetics of the experience. Testers mentioned that the breadcrumbs were unrealistic, but there are other aspects of the visuals that are also somewhat shoddy. The maze walls and floors could be made more interesting through the use of better textures and lighting, and the AI’s text box could have a better looking UI.

On a related note, the ambience of the game could be vastly improved via some simple additions. Changing up the size and contents of the rooms would really immerse the player more. I could also add discoverable objects, writing on the walls from past players, and parts of the maze that are dark or provide other interesting challenges. This feature could potentially be incorporated with the level or checkpoint design to maximize the effects of both.

There are more features that I have planned beyond just these – but these will be the focus of my attention for the short-term future.

**References and Resources**