ECE 594Q /CS 280 Introduction to Computer Graphics, Spring 2022

Homework 1: Maze Explorer

Due Monday, April 11, 2022 at 11:59pm (1 week)
(no late assignments allowed)

Introduction
In this homework, you'll write an OpenGL application that lets a user navigate a maze. Your application must do three things: 1) load the maze specification text file, parse it, and read in all the appropriate data into appropriate data structures, and 2) pass the geometry, texture information, etc. to the hardware and use OpenGL to render it, 3) handle the user input so they can navigate through the maze. In the last part of the assignment (Part 3) you will incorporate what you have learned in the first 2 parts to create your first "mini" OpenGL game. The idea is that in this homework assignment you will do everything in OpenGL, and in the next 3 assignments you will write a software rasterizer that will do this in software.

Part 1. Parsing the scene file and creating the maze
One of the things that comes up the most often in computer graphics is parsing scene data to load it into the rendering system. In a high-level framework like Unity this would be handled automatically for us, but for right now it is good for you to write at least one loader yourself. In this part of the assignment, you will need to write a parser that takes in the maze text file (see, e.g., maze*.txt) and creates the necessary data structures (lists of triangles) to prepare the system for rendering.

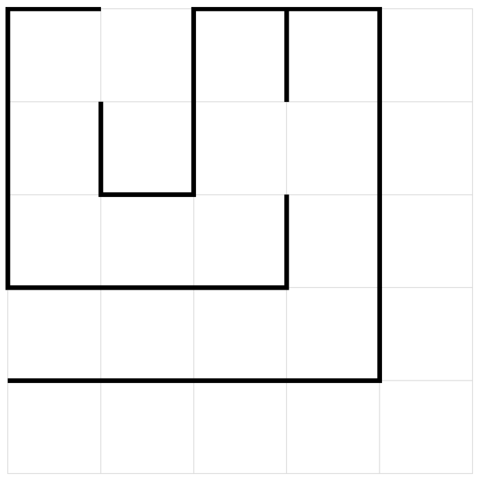
Maze file syntax
The maze scene file is a text file that describes what the maze will look like. **Anything on a line located after a pound sign (#) is considered a comment and should be ignored.** The basic building block of my maze is a "cell", which you can think of as a "room" in the maze. I use a FLOORPLAN to define which cells have walls and which ones don't in order to specify corridors, dead ends, etc.

To understand how this works, assume that we have a maze that has 5x5 cells. Assume that we have only one kind of wall (although your final implementation should allow for a user-specified number of different kinds of walls). Suppose the user specifies the following floor plan:

```
FLOORPLAN
 1   0   1   1   0
 1   0   1   1   1
 0   0   0   0   0
 1   1   1   0   1
 0   1   0   0   0
 1   0   0   1   1
 1   1   1   0   0
 0   0   0   0   1
 1   1   1   1   0
 0   0   0   0   0
```

Note that I could have spaced out these entries or put comments at the end of every line. Your parser should be able to handle this. In our syntax, a "0" indicates that there is no wall, while a 1 specifies that we use the wall defined by texture "1". In this example, since we have a maze of 5 x 5 cells, the FLOORPLAN array will be of size 10 x 5, because the rows alternate between defining the horizontal and vertical pieces of each of the cells. Your program should be able to read in this FLOORPLAN and generate a list of triangles that define the maze appropriately.

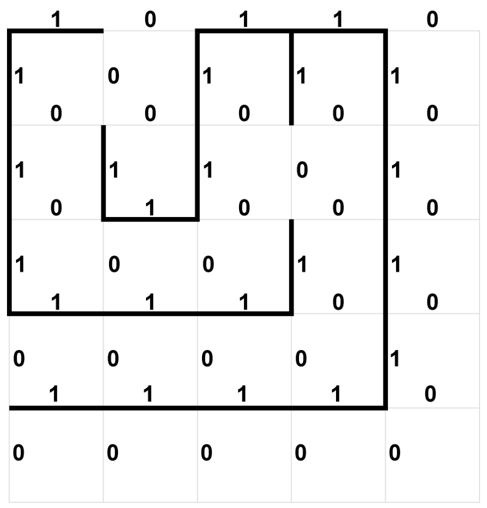
The maze that would be created by this FLOORPLAN would look as follows:



Note that I have defined this maze on a grid of 5x5 cells. In order to understand what is going on, remember that every other line defines the horizontal and vertical walls in alternating manner. In order to help me see what is going on, I can write the FLOORPLAN with spaces so that it looks as follows:

```
FLOORPLAN
   1    0    1    1    0
 1    0    1    1    1
   0    0    0    0    0
 1    1    1    0    1
   0    1    0    0    0
 1    0    0    1    1
   1    1    1    0    0
 0    0    0    0    1
   1    1    1    1    0
 0    0    0    0    0
```

All I have done when writing this is space out the entries a bit and offset every row from the other so that I can see which entries define horizontal walls and which ones define vertical walls. Note that I assume that the first row defines horizontal walls. I can use these 0's and 1's to label the walls of the map to help us understand their relationship to the final map:



So you can see how the FLOORPLAN defines which parts of the maze will have walls and which ones won't. In this case, I just have 1's and 0's, but in a more complex maze we can have many more numbers since we will have different kinds of walls (basically the texture changes on each wall, and every wall has a texture). I chose this format because it was easy to define fairly arbitrary mazes in this fashion.

Once we understand the basic format for describing the maze, let's talk about the syntax of the file. The maze specification file consists of a series of COMMANDS (written in ALL CAPS), followed by specific attributes. The commands that must appear in the file (and that you should be able to parse) are:

`DIMENSIONS [cells wide] [cells high]` – this specifies the dimensions of the maze, in cells. In the example above, we would have entered:

```
DIMENSIONS 5 5
```

`HEIGHT <height>` - this is the height OpenGL should use for the triangles. You should assume that the *(x,y)* coordinates of every point define its position on the floorplan of the maze, and the *z* coordinate is the height off the ground. Therefore, the floor of the maze should be positioned at z = 0.

CELL <size> - this defines the size of a cell in world coordinates. The cells of the maze are square, and usually I make the cell size equal to the height of the walls (making every wall of a cell a perfect square) but they don't have to be. So essentially CELL and HEIGHT will help you define the coordinates for the vertex of every triangle of every wall.

TEXTURES <number> followed by a list of
<count> <tex filename> - This defines the textures that are to be used for the walls of the maze. In my maze1.txt example, I use 9 different wall textures. You will need to read in these textures and use them to texture map the walls of the maze. I use numbers 1 – 9 to specify each, and then in the FLOORPLAN I define where on the maze these walls will

appear. Note that in my file format, the last texture (in maze1.txt it is texture #9) defines the floor texture.

Note that the DIMENSIONS, HEIGHT, CELL, and TEXTURES commands can come in any order in the file. Let's assume for this homework that the FLOORPLAN command comes at the end, once the DIMENSIONS are known.

In my worldspace coordinates, the origin (0,0,0) is on the floor at the center of the maze.

Notes:

- You can build a pair of triangles for every wall of a cell, you do not need to combine several wall segments into a single wall. This is not as efficient but will make things easier in constructing the maze.

- The walls of the maze can be infinitely thin, meaning they have no thickness

- You will need to define texture coordinates at every vertex so that you can texture map the surfaces correctly.

- Make the floor of the maze of size `cell_size * cells_wide` by `cell_size * cells_high`. Position the floor at z = 0 and make sure you texture map it with the last texture identified. Make the floor one large quad, but texture map it so that you "tile" the texture across it (as opposed to stretching it over the entire floor).

- Walls can be single-sided, meaning you don't have to draw two sets of triangles (one front-facing, one back-facing) to cover the wall. This means that the textures will be backwards in certain views of the wall but that's okay.

- **Make sure you write your own parser**. Do not "borrow" from other parser systems online, or use lex/yacc, etc. If you do not know how do to basic I/O from a text file, then learn how to do so from an introductory C book.

Part 2. Drawing the maze with OpenGL and handling user input

Once you are able to read in the maze file and construct the necessary data structures (i.e. a list of all the triangles of the scene), you will need to hand them off to OpenGL to draw them. You can use the immediate mode for rendering triangles, just make sure that you also pass in the appropriate texture coordinates so that the textures can be mapped. Remember you should have also drawn the floor correctly.

Use GLUT to handle the user interaction. You can emulate the interaction I have in my demo example, where I use the mouse to move the user's viewpoint and I use the w,a,s,z keys to move the user around the maze by changing the user's *(x,y)* position. Call the resulting program MazeViewer.exe. Make sure that it takes as an argument the maze specification text file. Also make a new maze by creating a new maze specification that you will submit along with your code. If you have new images for the walls, please include them as well.

Submit the results of Part 1 and Part2 together. To grade this part of the assignment, I will test your code and compare it with my own to make sure that it works correctly. Before you start, it may be a good idea to try out my sample executable on maze1.txt and maze2.txt (which is the maze example we talked about here) to see how it works.

Part 3. Maze Explorer game

Once you are able to draw the maze correctly and handle input, you will make a simple maze game, where the user starts at the center of the maze and has to get out before the time expires. To do this, first create some interesting maze that has only one exit. Start off the user in the center of the maze, and give them a fixed amount of time, shown on a timer that counts down (say 1 minute). Remember to implement collision detection so that the user cannot walk through the walls! You will also have to modify the view control that I had in MazeViewer so that the user can properly walk along the maze. End the game when the user reaches the exit (or when the time runs out) by showing the appropriate graphic on the screen. At this point, you will have finished your first OpenGL game, congratulations!

Extra credit

This assignment has plenty of room for extra credit. Examples include:

- Adding enemies (don't have to be articulated) that move around the maze to chase down the player. The game ends if the player gets caught.

- Add more realistic lighting to the walls of the maze.

Final words

This assignment is due in a week, so please get started right away**. Because I will release the solutions for this assignment right after the deadline, no late assignments will be allowed for this assignment!** So make sure you get done in time.

Implement the parsing first, and make sure that you are creating the appropriate geometry for the scene. Then start rendering it with OpenGL. If you are starting to get familiar with OpenGL, that part will actually be quite easy. Finally, implement the game mode. My expectation is that you will spend 40% of your time with Part 1, 30% with Part 2, and 30% with Part 3, approximately.

Get started soon, and post questions in the class forum if you get stuck. Zip up the two main projects (part 2 and part3) and submit them through GauchoSpace in a single zip file with your name on it. If you are using CSIL as the environment for development, please provide the Makefile and all the files needed in your submission. Good luck!