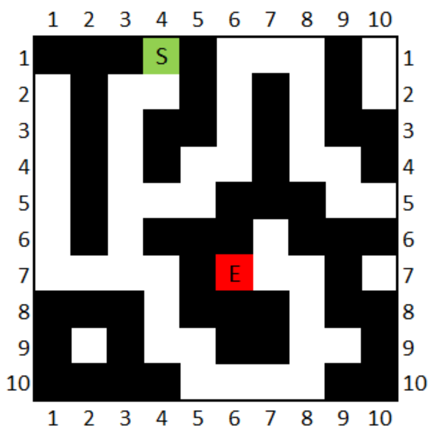


24-780 B—Engineering Computation

Assigned: Thurs. Sept. 29, 2022
Due: Tues. Oct. 11, 2022, 11:59pm

Problem Set 5: 2D Maze

Before anything else, a clarification:

> *This assignment is the first part of a two-part sequence. For PS05, we will deal simply with drawing the maze as it is read from a file, with some limited user interaction. For PS06, we will deal with how to implement an algorithm that can allow an entity to navigate the maze automatically. Again, no automatic navigation yet!*

For this assignment, you are asked to write a program that can read information about a maze from a data file and display the maze in an OpenGL graphics window. Your program will also draw an entity that a user can manipulate to navigate the maze. I recommend that you tackle this assignment by progressing through the tasks listed, being sure each part works before continuing.

The Maze

The maze is a two-dimensional map made up of squares, each with dimensions of 20x20 pixels. Each square is referenced by row and column number, starting with 1 (e.g., the top left square is 1,1). Each square is either *navigable* (represented in white) or *obstacle* (represented in black). In addition, there is a navigable square that represents the starting point of the navigation challenge and there is a navigable square that represents the end point (i.e., the goal) of the navigation challenge (see sample above).

Although the maze can have any dimension in rows and columns, you may assume that the maze will never be larger than 40 squares on one side.

The configuration of the maze is read in from a text file that has a format as given at right. The start and end locations are given as rowNum followed by colNumb (like for a matrix). The map is defined first with the key words “Map Begin:” followed by the number of columns (10 in this example). The rest of the lines (up to key words “Map End”) give the location of obstacles (represented by “1”) and navigable spaces (represented by “0”). Number of rows in the maze is determined by how many lines are used to define the map. Note that Start and End could be given before or after map definition (or omitted completely).

```
Start: 1 4
End: 7 6
Map Begin: 10
1 1 1 0 1 0 0 0 1 0
0 1 0 0 1 0 1 0 1 0
0 1 0 1 1 0 1 0 1 1
0 1 0 1 0 0 1 0 0 1
0 1 0 0 0 1 1 1 0 0
0 1 0 1 1 1 0 1 1 1
0 0 0 0 1 0 0 0 1 0
1 1 1 0 1 1 1 0 1 1
1 0 1 0 0 1 1 0 0 1
1 1 1 1 0 0 0 0 1 1
Map End:
```

The example shown generates the maze illustrated above. This input file and other (bigger) ones are attached to this assignment to use for testing.

Task 1 (*Maze* class)

Create a class for the maze called *Maze*. Think about what data needs to be stored and what is the best method for doing so. What member variables are needed? What functions make sense and should they be private or public? Don’t overcomplicate things (e.g., using a 2D vector where a 2D basic array will do, or using integer where boolean is more appropriate).

Reading a file

`void readFile(std::ifstream &input);`

Create a function that, when given an input file stream, *clears* the maze and then reads the file to get all maze data. Since we don't want to just read blindly (we need to check for keywords), you need to make this function "a little bit smart". Here is an example of the kind of code you need to accomplish this, with an example for reading the start location of the maze:

```
        string wholeLineString; // used to read file one whole line at a time
        stringstream lineStream; // used to easily get numbers from part of line
        int colonLocation; // used to store location of colon in line

        while (!input.eof()) {
                lineStream.clear();// just in case

                getline(input, wholeLineString); // read the whole line into a string

                // if there is a colon, set up lineStream to start just after colon
                // also, remember colonLocation, just in case
                if ((colonLocation = wholeLineString.find(":")) != string::npos)
                        lineStream.str(wholeLineString.substr(colonLocation + 1));
                else {
                        colonLocation = -1;
                        lineStream.str(wholeLineString);
                }

                // check for keywords in the line
                if (wholeLineString.find("Start:") != string::npos) // find() returns npos when not found
                        partLineStream >> startRow >> startCol; // put values into class variables
```

You decide how else to work this function to get the data into your maze. For instance, *lineStream* may sometimes contain the whole line (no colon) and will be read in some kind of loop.

Writing the maze

`friend ostream& operator<<(ostream& os, const BitmapMaze& aMaze);`

This function outputs all the maze data to an output stream (file or console) so that it looks just like the input file. For Task 1, the purpose of this function is purely so that you can be sure that you captured all the information correctly and can do it for any maze input file.

Create a simple/short *main()* function (in a separate file) to test the functioning of your program up to Task 1 (reading and writing mostly). You can have your program ask for an input file (with the extension .map) or hardwire the different file names.

Task 2 (OpenGL, no interaction)

Write a function *paint(...)* that displays the maze graphically in OpenGL. Note that maximum size of 40x40, with 20 pixels each square gives you a window size of 800x800. You are NOT required to include the index numbers along the edge of the maze (keep it simple). The Start Square should be colored green and the End Square should be colored red. The inclusion of the letters "S" and "E" is optional. Pressing the ESC key should close the window.

Again, use *main()* to test the functioning of your program up to Task 2, using different size mazes from the input files provided. Note that the maze being drawn may not fill the whole window. The unused squares should just be grayish.

Task 3 (User-controlled navigation, *Entity* class)

Create an *Entity* class that the user can manipulate through the maze by using the arrow keys. The entity should be smaller than 20x20 pixels (so that it fits inside each square) but can appear to be anything (e.g., a circle, a spider, a robot, an alien, a letter, etc.). Obviously, the entity cannot move into a square that is not navigable (the square is an obstacle, or the square is beyond the maze area). It is very likely Entity class will have functions like:

- `void move(Maze &aMaze, int direction);`
- `void paint();`
- `bool reset(Maze &aMaze); // places entity at maze start`
- `bool reachedGoal(Maze &aMaze);`

The variable "direction" will probably use the same constants as fsSimpleWindow (e.g., FSKEY_UP, FSKEY_LEFT, etc.), so there is no need to enumerate the four possible moves.

Maze class will likely now require some new functions like:

- `bool isNavigable(int row, int col);`
- `Cell getStart();`
- `Cell getEnd();`

Note that *Cell* is a struct with two integer members, *row* and *col*. This struct is defined in Maze.h, but is not a part of the *Maze* class itself.

Append your *main()* function to manage the additional key entries (which includes reset) and to show the location of the Entity in the maze. You may need a "menu" in the console at this point.

Feel free to "celebrate" the user if/when they reach the goal. Although NOT required, you may provide feedback on number of steps taken, or maybe how much time it took to navigate the maze . . .

At this point, you've created a nice little game that you and your friends can play. Have fun!

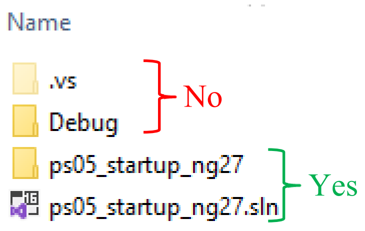
Deliverables

3 files, very appropriately named and zipped together:

PS05maze_main_andrewID.cpp		<< contains *main()* and maybe some other code
Maze.h	**Maze.cpp**	<< contains Maze class
Entity.h	**Entity.cpp**	<< contains Entity class

Upload the zip file to the class Canvas page before the deadline (Tuesday, Oct. 11, 11:59pm).

Alternatively, if you are using Visual Studio, it may be easier to submit your entire solution rather than a collection of files. To do this, create a *zip file* of the whole project (the .sln file and the associated folder), being careful NOT to include the hidden folder called ".vs". This folder is used only to manage the IDE and is typically huge (>100MB). Erasing or omitting it will just force Visual Studio to rebuild it when needed. The Debug folder should also be kept out of the zip file to avoid including executable files that some firewalls may disallow. *The name of the project should include your AndrewID*



Learning Objectives

Read data from *structured* input files.

Develop your own classes and the interactions between them

Providing user interaction.

Searching references (online or textbook) for C++ library functions.