CS559 Spring 2001
Project 2
Maze Visibility and Rendering **Due Friday April 13**

Your task in this project is to implement a maze rendering program, not too far removed from those used in computer games of the first-person shooter variety. Read this entire document carefully before beginning, as it provides details of the required implementation and various tips.

1 Mazes

A maze consists of rectangular cells separated by edges. The edges may be either transparent or opaque. The viewer is supposed to see through transparent edges into the neighboring cells, and they should not see through opaque edges. Each edge is assigned a color (which is meaningless for transparent edges).

The maze is described as a 2D structure **assumed to lie in the XY plane**. To make it 3D, each edge is extruded vertically from the floor to the ceiling. **The floor is at $\mathbf{z = -1}$ and the ceiling is at $\mathbf{z = 1}$.** Each wall should be drawn with its assigned color.

Associated with the maze is a viewer. The viewer has an xy location, a viewing direction, and a horizontal field of view. The view direction is measured in degrees of rotation about the positive z axis. The horizontal field of view is also measured in degrees. **The viewer is assumed to be at $\mathbf{z = 0}$.**

The maze file format consists of the following information (also look at one of the example mazes):

- The number of vertices in the maze, n_v. Each edge joins two vertices.
- The location of each vertex, specified as it x and y coordinates. The vertices are assumed to be numbered from 0 to $n_v - 1$.
- The number of edges in the maze, n_e. Remember, there is an edge between every cell, even if that edge is transparent.
- The data for each edge: the index of its start vertex, the index of its end vertex, the index of the cell to the left, the index of the cell to the right, a 1 if the edge is opaque, or 0 if transparent, and an RGB triple for the color. The left side of an edge is the side that would appear to be on your left if you stood at the start of the edge and looked toward to end. If there is no

cell to the left or right, an index of -1 is used. The edges are assumed to be numbered from 0 to $n_e - 1$.

- The number of cells in the maze, n_c.
- The data for each cell, which consists of the four indices for the edges of the cell. The indices are given in counter-clockwise order around the cell.
- The view data, consisting of the x and y viewer location, viewing direction and the horizontal field of view.

2 Software Provided

Several classes have been provided. Together they build to two programs. The first program creates mazes in a certain format. The second is a skeleton maze renderer. The code is reasonably well documented, but part of the project is figuring out how the given code works and how to integrate your code into it. The programs are described below. To build them, set the appropriate active configuration in Visual C++ and build.

2.1 BuildMaze

The BuildMaze program provides a simple user interface for building mazes. The user specifies the following parameters:

Cells in X: The number of cells in the x direction.

Cells in Y: The number of cells in the y direction.

Cell X Size: The size of the cells in the x direction.

Cell Y Size: The size of the cells in the y direction.

Viewer X: The initial x location of the viewer.

Viewer Y: The initial y location of the viewer.

Viewer Dir: The initial viewing direction, given in degrees of rotation about the positive z axis (the standard way of specifying a rotation in the plane).

Viewer FOV: The horizontal field of view of the viewer.

The Build Maze button builds a maze with the given parameters and displays it. The Save Maze button requests a file name then saves the maze to that file. The Load Maze button requests a maze file to load and display. Quit should be obvious.

2.2 RunMaze

The RunMaze program provides a skeleton for the maze walkthrough that you will implement. As provided, it displays both a map of the maze and an OpenGL window in which to render the maze from the viewer's point of view. On the map is a red frustum indicating the current viewer location, viewing direction and field of view. The map is intended to help you debug your program by indicating what the viewer should be able to see.

To move the viewer, hold down the left mouse button and drag **in the OpenGL window**. Mouse motion up or down is translated as forward or reverse motion of the viewer. Left and right mouse motion changes the direction of view. As the skeleton stands, the viewer will move in the map window to reflect the mouse motion.

The system performs collision detection between the wall and the viewer to prevent the viewer from passing through opaque walls. You should examine the code that does that to see an implementation of Liang-Barsky clipping (in essence). The RunMaze program also keeps track of which cell the viewer is currently in, which is essential information for the cell-portal visibility algorithm you must implement.

You should pay particular attention to the function draw in MazeWindow.cpp that sets up the OpenGL context for the window. As you will read later, all of the drawing you do in this project must be in 2D, so the window is set up as an orthogonal projection using the special OpenGL utility function gluOrtho2D. That function also draws the projection of the ceiling and the floor of the maze. You should be able to reason as to why is it safe to treat the floor and ceiling as infinite planes (hint: the maze is closed), and why those planes project to two rectangles covering the bottom and top half of the window.

2.3 C++ Classes

This document will not go into details of the C++ classes provided. You should spend a considerable amount of time perusing them to figure out how everything works, and too look for little functions that will be useful in your implementation,

such as functions to convert degrees to radians and back again (recall that all the C++ trigonometry functions take radians).

3 Your Task

Produce the viewer's view of the maze. You must extend the function Maze::Draw_View to draw what the viewer would see given the maze and the current viewing parameters. Note that the function is passed the focal distance, and you also have access to the horizontal field of view. Your implementation must have the following properties.

- You must use the Cell and Portal visibility algorithm to achieve exact visibility. In other words, apart from drawing over the floor and ceiling, no pixel should be drawn more than once. The algorithm is given in psuedocode below.

```
Draw_Cell(cell C, frustum F)
    for each cell edge E
        if E is opaque
            E' = clip E to F
            draw E'
        if E is transparent
            E' = clip E to F
            F‘ = F restricted to E'
            Draw_Cell(neighbor(C,E), F')
```

  The function Draw_Cell(C, F) is initially called with the cell containing the viewer, and the full view frustum. The neighbor(C,E) function returns the cell's neighbor across the edge. Note that drawing a 2D edge means drawing a wall in 3D. For an example of the algorithm in action, look back as the slides for lecture 15.

- You are only allowed to use OpenGL 2D drawing commands. In other words, any vertices you specify should use glVertex2f, glVertex2fv, glVertex2d or glVertex2dv only. You should use glBegin(GL_QUADS) to draw quadrilaterals, and glColor3f or glColor3fv to specify the polygon color. **We will check for other OpenGL calls when we grade.**

- As an side effect of the 2D restriction, you must do your own viewing transformation. That is, you must take points specified in world space (where you will do the visibility) and transform them all the way into screen space (where you will draw them.) The transformation will consist of a translation and rotation to take the points from world to view space (with the origin at the viewer's location) and then a perspective division to take the view space points into screen space. Note that you are given the focal distance to make things easier, but you must still take care of several small details.

4 Helpful Tips

- The visibility algorithm is a 2D algorithm in this case, because all the walls are vertical and the viewer is looking horizontally. That also means that all the pieces of wall that you draw will have vertical left and right edges. They will not have horizontal top and bottom edges due to perspective effects.

- Implement a Frustum class that stores information about a viewing frustum, and has a method for clipping a frustum to an edge.

- Implement a function in the Edge class or the LineSeg class that clips an edge to a given view frustum. You will have to work out a way to compute the intersection point of a line segment with an infinite line in 2D space. Start by writing out the equations of the lines in parametric coordinates. There is a function in the LineSeg class that may help get you started.

- It is easiest to begin with a 1 by 1 maze, in which case there is no recursive step. That gives you the opportunity to debug the transformations and projection before getting into the details of manipulating view frustums.

- There are lots of interesting extensions for this project, including lighting and texture mapping. They are deceptively difficult to implement in the context of this project due to the way the floor, walls and ceiling are drawn. **Do not include any extensions in the program you submit**. If you wish to experiment I recommend stepping outside the 2D restriction of this project.

5 Grading and Submission

The project will be graded out of 50, of which you get 5 points just for having a program that compiles and runs without crashing. There are no optional parts to this assignment, although you are welcome to experiment with extensions. **Do not submit any extensions.**

Submission and demo-based grading will work similar to project 1. More details will follow.