Project 2 - Viewing

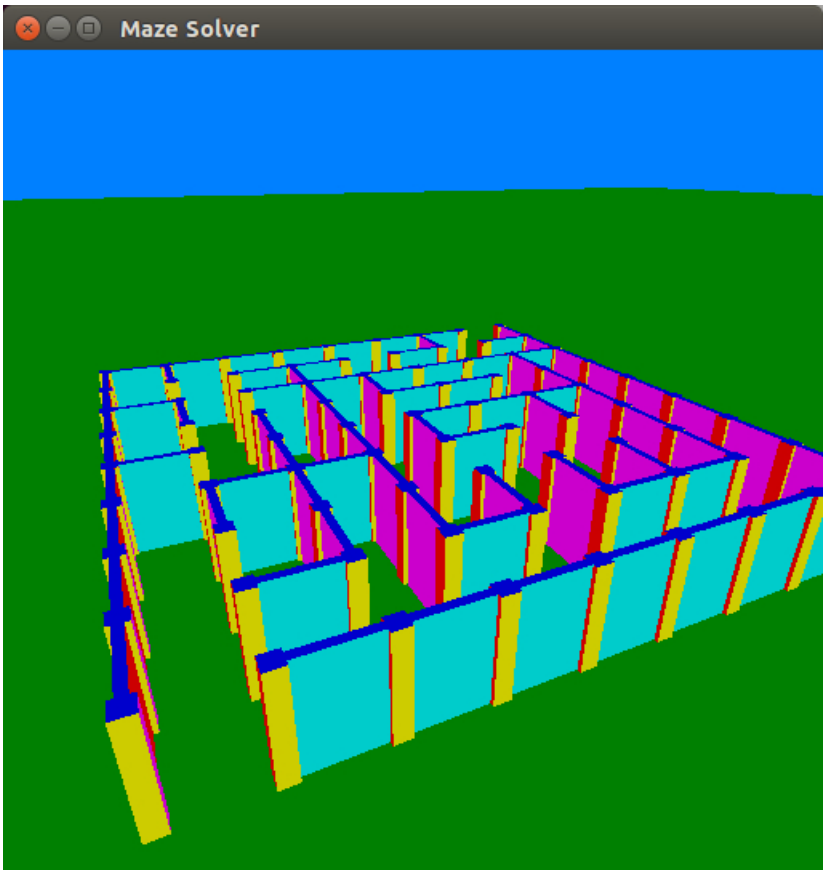
CS 1566 — Introduction to Computer Graphics

Check the Due Date on the CourseWeb

The purpose of this project is for you to familiar with model view matrices and projection matrices.

Generate a World (Object) Frame

For this project, your world (object) frame will contain an 8 by 8 maze as shown below:



To create this world frame, first you need to generate an 8 by 8 maze. The following three files are given to help you generate a maze:

- `genMaze.h`: The header file for `genMaze.c`
- `genMaze.c`: The implementation file

- `genMazeExample.c`: An example of how to use `genMaze.h` and `genMaze.c`

To use the above program, first include `genMaze.h` into your application. Next, create a variable of type one-dimension array of `cells`. Since we are going to create 8 by 8 maze, you can use either

```
cell cells[64];
```

or

```
cell *cells = (cell *) malloc(sizeof(cell) * 8 * 8);
```

to allocate the variable. Note that the type `cell` is defined as follows:

```
typedef struct
{
    unsigned char north;
    unsigned char east;
    unsigned char south;
    unsigned char west;
} cell;
```

Each `cell` consists of four components, `north`, `east`, `south`, and `west`. Each component will be either 0 or 1 after the maze is generated by the `genMaze()` function. If the `north` component is 0, it means there is no wall in the north direction of that cell. If the `north` component is 1, there is a wall in the north direction. Meaning are the same for `east`, `south`, and `west` components for east, south, and west direction of the cell.

To generate an 8 by 8 maze, simply call the function `gen_maze()` as shown below:

```
gen_maze(8, 8, cells);
```

The first argument is the number of rows and the second argument is the number of columns (8 by 8 in this case). If you want to work with two-dimensional array instead of one-dimensional array, you can cast the variable `cells` to two-dimensional array once the maze is generated using the following statement:

```
cell (*cells2D)[numColumns] = (cell (*)[numColumns]) cells;
```

Note that you have to replace the variable `numColumns` by 8 in this case. After casting, you will be able to use the variable named `cells2D` as a two-dimensional array of `cells`. For example, if you want to view the value of the component `north` of the cell at row 2 column 5, simply use the following statement:

```
printf("%i\n", cells2D[2][5].north);
```

The `print_maze()` and `get_num_walls()` functions are also provided. The `print_maze()` function will print a maze in a text mode on the console screen. The `get_num_walls()` function will return the number of walls of a given maze. This will allow you to setup your array of vertices correctly.

Placing Objects into World Frame

Once you generate a maze, you can use it to help you place objects (walls) into your world frame to make them look like a maze. A wall is simply a cube that get flatten by scaling. Make sure to use different color for each side of a wall. In the example above, all sizes of walls that face the same direction have the same color. So, to help distinguishing between two walls that face the same direction, you can put a pole in between as shown in the example. To distinguish between the ground and the sky, you can put a huge cube as a ground as shown in green color in the above example.

Viewing Your World

For this project, we are going to view our maze using the perspective projection as discussed in class. Thus, you must implement the following functions:

```
mat4 look_at(GLfloat eyex, GLfloat eyey, GLfloat eyez,
             GLfloat atx, GLfloat aty, GLfloat atz,
             GLfloat upx, GLfloat upy, GLfloat upz);

mat4 frustum(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top,
             GLfloat near, GLfloat far);
```

These matrices will be sent to the graphic pipeline to change the frame (model view matrix) and projection (projection matrix). Thus, your vertex shader file (`vshader.glsl`) should look like the following:

```
#version 130

in vec4 vPosition;
in vec4 vColor;
out vec4 color;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

void main()
{
   color = vColor;
   gl_Position = projection_matrix * model_view_matrix * vPosition / vPosition.w;
}
```

or

```
#version 120

attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;
```

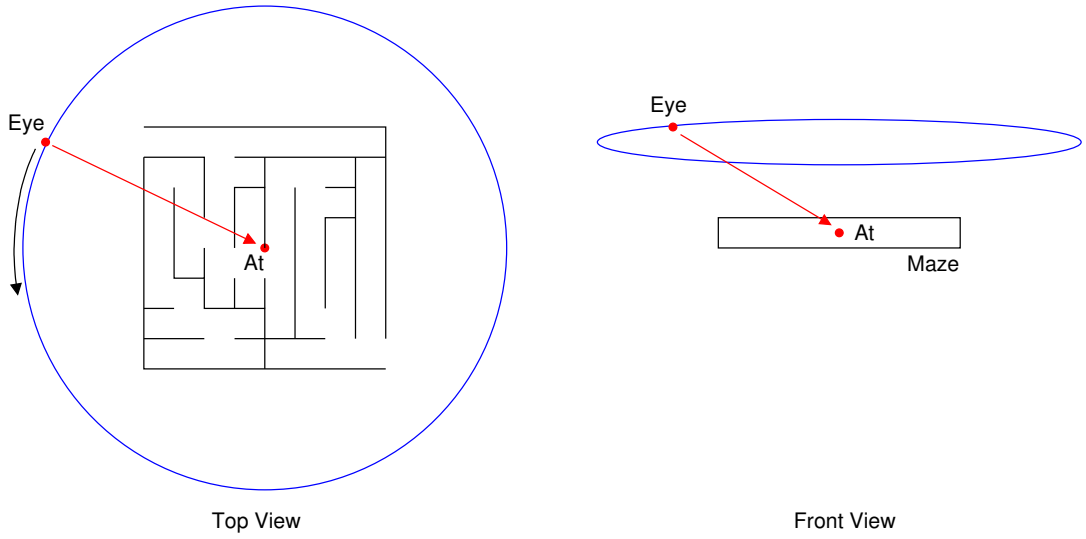
```
uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

void main()
{
   color = vColor;
   gl_Position = projection_matrix * model_view_matrix * vPosition / vPosition.w;
}
```

In your application, you have to locate two variables `model_view_matrix` and `projection_matrix` in the same way as in project 1. Note that in your `display()` function, you need to send two matrices for this project. The vertex shader program (`fshader.glsl`) should be the same as in the project 1.

Task 1: Flying Around the Maze

The first task is to fly around the maze. What you need to do is to pick an eye point that is a higher than the maze and look at the center of the maze. Then simply move your eye point around the center of the maze for 360 degree as shown in a top view of a maze below:



So, pick an eye point (preferably on the same side of an entrance) and simply move the eye point around as shown above.

Task 2: Flying Down

After the first task is finished, the next step is to fly down to the front of the entrance. For this task, simply slowly change the eye point from the final position of the task 1 to the front of the entrance. At the same time, you also need to slowly change the at point from the center of the maze so that you will look straight into the maze once the eye point is right at the front of the entrance.

Task 3: Solving the Maze

The last task is to solve the maze and make it like a person is actually walking into the maze. You can use any maze solving algorithm (left-hand rule or backtracking with recursion). To simulate waling into the maze, what you need to do is to repeatedly change eye point and at point. Once you at the exit, simply stop.

HINTS

For this project, there is a lot of animation but all of them are nothing but slowly changing eye point or at point or both. This should be done in `idle()` function. One trick to smoothly change from one point to the other is to use point-vector addition. For example, suppose you want to change from point P_1 to point P_2. This can be done by

$$P2 = v + P_1$$

where $v = P_2 - P_1$. To slowly change it, simply adjust the magnitude of v using the following formula:

$$P2 = \alpha v + P_1$$

where α slowly changing from 0 to 1.

Submission

The due date of this project is stated on the CourseWeb. Late submissions will not be accepted. Zip all files related to this project into the file named `project2.zip` and submit it via CourseWeb. After the due date, you must demonstrate your project to either TA or me within a week after the due date.