Project 2 - Viewing

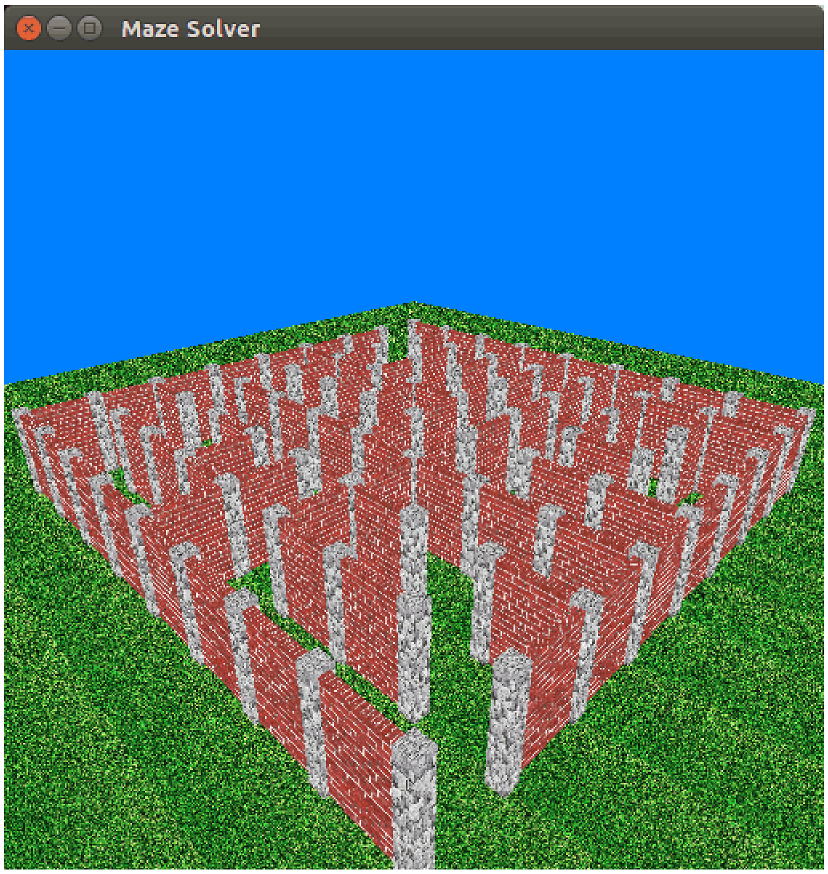
CS 1566 — Introduction to Computer Graphics

Check the Due Date on the Canvas

The purpose of this project is for you practice creating a complicate world (a maze) with textures and 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 a data structure that represents an 8 by 8 maze. An example is a two dimensional array size 8 × 8 of `cells` where a cell is a structure as shown below:

```
typedef struct
{
    int north;
    int east;
    int south;
    int west;
} cell;
```

Each `cell` contains four components, `north`, `east`, `south`, and `west`. Each component can be either 0 or 1 that represents the existing of a wall. For example, suppose we declare the variable named `cells` as follows:

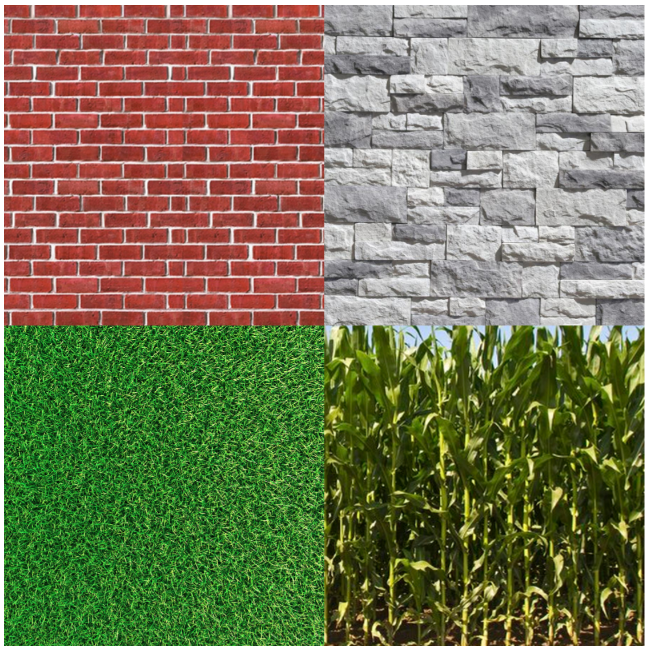
```
cell cells[8][8];
```

If `cells[2][5].north` is 1, it means that the cell at row 2 column 5 contains the north wall. If `cells[3][6].west` is 0, it means that the cell at row 3 column 6 has no west wall. This is just an example of a data structure that can be used to represent a maze. **You do not have to use this data structure. You can come up with your own data structure if you want but you have to make sure that it will allow you to create a maze 3D world and navigate (solve) it.**

The maze should be randomly generated every time you run your program. From a location (cell) in the maze, there must be exactly one solution (path) to another location in the maze. If you generate your maze manually and static (same maze every time you run your program or your maze is not a good maze, some points will be deducted.

Texture

For this project, a texture is given as shown below:



The above image is an 800 × 800 pixels. The raw image file (`p2texture04.raw`) is given on the Canvas under this project. The above image contains four textures as shown above. Note that you do not have to use this texture. **You can create your own image containing at least 3 textures. For this project, you must use at least three textures, one for walls, one for poles, and one for the floor**.

Placing Objects into World Frame

Once you generate a maze, use it to guild you to construct a maze in 3D world. Your maze data structure should help you identify location of walls and poles. **It may be a good idea to construct your maze on a plane** $y = 0$ **or** $y = -0.5$. This will make sure that the **up** vector will be $(0, 1, 0)$.

In our 3D world, a wall is simply a cube that get flatten in one direction by scaling. In the example above, all sides of a wall use exactly the same texture (bricks) for simplicity. Similarly, a pole is a cube that get flatten in two directions by scaling. All sides of a pole use exactly the same texture (stone). The floor is just a bunch of cubes that have been flatten to a thickness with the grass texture.

Note that once your program finish constructing the maze, the maze will remain unchanged throughout the program. Because of this, you should construct and finalize your maze in your application. In other words, you will have a single array of vertices which contains a number of walls and poles. Each object has been transform into its final position and orientation. In doing so, it allows you to use the `glDrawArrays()` function only once to draw the whole maze without calling `glUniform4fv()` mutliple times to set `ctm` for each cube.

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 ortho(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top,
           GLfloat near, GLfloat far);

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

For this project, you must use the perspective projection (frustum). However, the orthographic projection may help you ensure that you construct your maze correctly. It is harder to debug with frustum since it is more sensitive and hard to imagine what you should see.

Note that instead of having 9 arguments, the `look_at()` function can have just three arguments as shown below:

```
mat4 look_at(vec4 eyePoint, vec4 atPoint, vec4 upVector);
```

So, it is up to you whether you want to use which signature.

Recall that these matrices will be sent to the graphic pipeline to change the frame (model view matrix) and projection (projection matrix). Also, we are not going to use vertex colors in this project. So, we do not need to generate and send the array of colors as we generally do. We also need to use the texture which requires texture coordinate. Thus, your vertex shader file (`vshader.glsl`) should look like the following:

```
#version 120

attribute vec4 vPosition;
attribute vec2 vTexCoord;

varying vec2 texCoord;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

void main()
{
   texCoord = vTexCoord;
   gl_Position = projection_matrix * model_view_matrix * vPosition;
}
```

Note that the above `vshader.glsl` does not have the `ctm` (current transformation matrix). If your project needs a `ctm`, feel free to incorporate it into your project.

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, model view and projection matrices. The vertex shader program (`fshader.glsl`) should be the same as in the texture lab:

```
#version 120

varying vec2 texCoord;

uniform sampler2D texture;

void main()
{
        gl_FragColor = texture2D(texture, texCoord);
}
```

Animation

There are a number of animations in this project. Make sure to utilize the `idle()` function as a mean of repeated operations or animation. See the set of slides about animation posted under a page or with this project on Canvas.

What to do?

For this project, we are going to grade your result based on the following tasks:

1. **Dynamically Generate Mazes (20 Points)**: Note that if you are going to generate a maze every time your program is executed, your maze should be a good maze. In other words, there must be exists exactly one solution from one location in the maze to another. **If your maze is static (same maze every time), you will get no points for this part.**

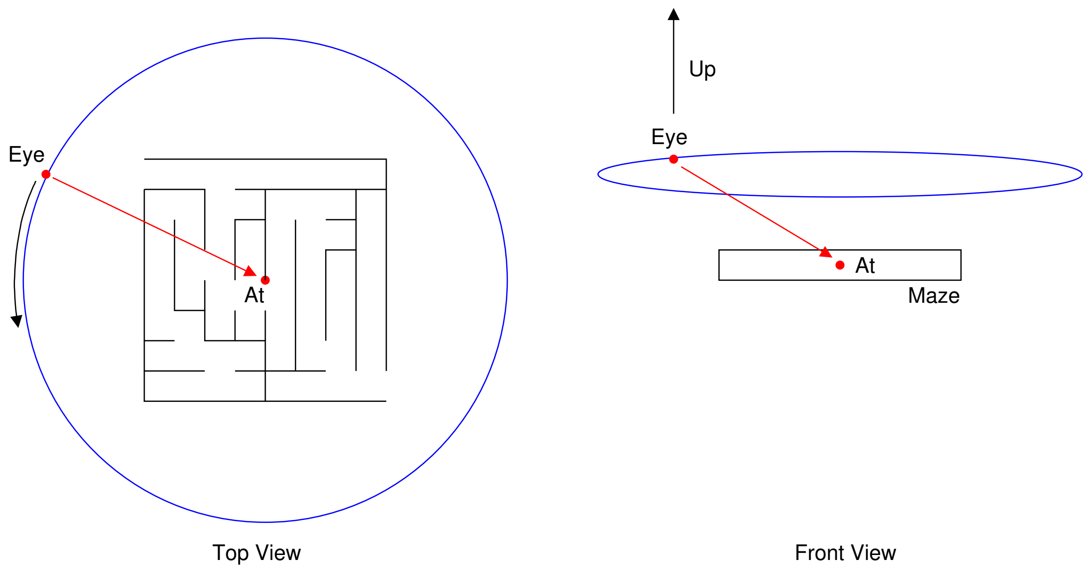
2. **Generate the World (40 Points)**: Use the maze that you generate from the first part to generate a 3D world that consists of nothing but transformed cubes. If you want, you can use more complicate objects to construct your world. For example, use a cylinder as a pole. Your object must use texture(s) instead of colors. If your maze does not use texture (only use solid colors), you will only get at most 20 points for this part.

3. **View from the Top (10 Points)**: When your program is executed, the first thing that it should show is the whole maze from the top view. This will allow us to verify the following:

   - Your maze is a good maze
   - Your program generates a new good maze every time it is executed

   At this point, your program should wait until the user press a key (e.g., SPACE). Once the user press the key, create an animation that simulates a flying from the top of the maze to one of the corner (for the next task) while looking straight at the center of the maze.

4. **Flying Around the Maze (10 Points)**: The purpose of this 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. This must be the same point from previous task. 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.

5. **Flying Down (10 Points)**: After the previous 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 previous task 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.

6. **Solve the Maze (30 Points)**: This task is to solve the maze and make it looks like a person 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 exit the maze, turn around 180 degree.

7. **Shortest Path (30 Points)**: Once you get out of the maze from the previous task, the next step is to go back to the starting point of the maze. **For this task, you must use the shortest path**. The shortest path can be found using a various method. One easy way is to remember all locations that you visit from previous task (solving) and optimized it (removing all location in between two duplicates and the duplicate). Once you are back at the entrance, 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 Canvas. Late submissions will not be accepted. Zip all files related to this project into the file named `project2.zip` and submit it via Canvas. You must demonstrate your project to either TA or me before the due date or on the due date during your recitation sessions.