Project 2 - Viewing

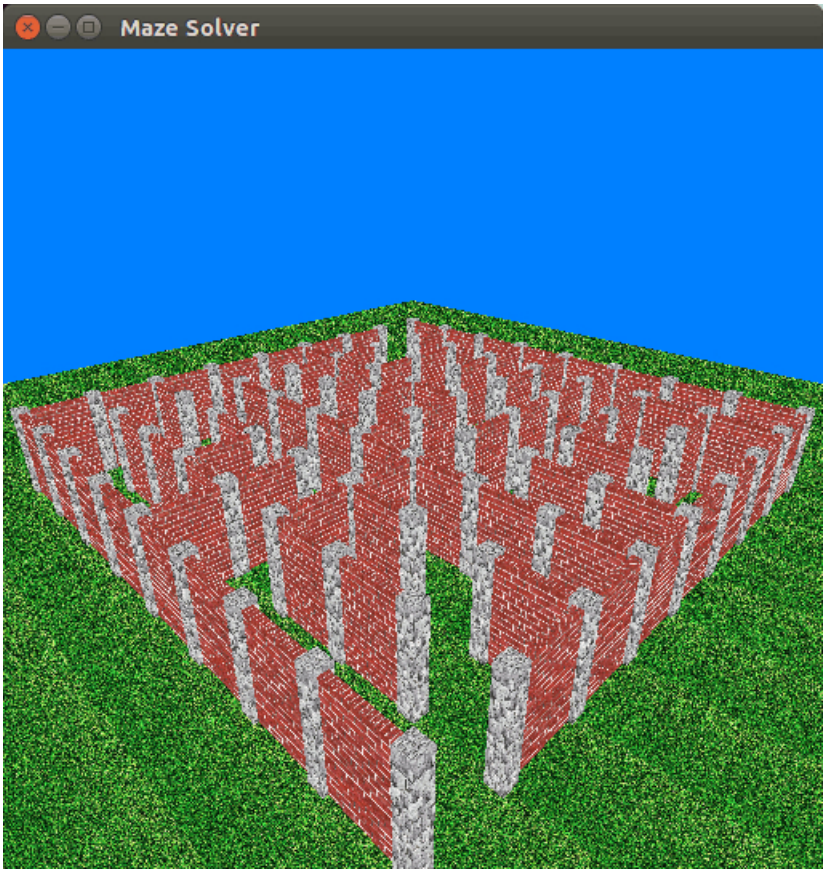
CS 1566 — Introduction to Computer Graphics

Check the Due Date on the CourseWeb

The purpose of this project is for you practice creating a complicate world (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:



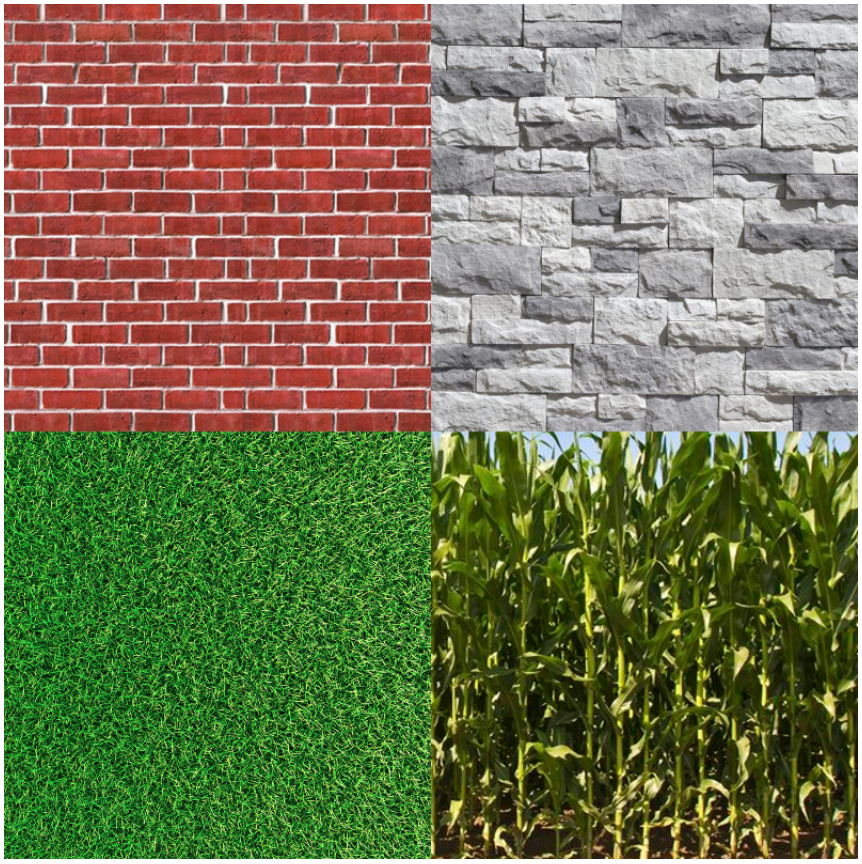
To create this world frame, first you need to generate an 8 by 8 maze.As we discussed in class, you have two choices:

1. Manually create a static maze (same every time your program is executed) (-10 points)
2. Generate a new maze every time the program is executed

Note that you maze should be a kind of data structure that can be used to generate your world.

Texture

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 CourseWeb under this project. Note that you do not have to use this texture. You can create your own texture as long as you have 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, you can use it to help you place objects (walls and poles) into your world frame to make them look like a maze. 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 with the grass texture.

As I mentioned in class, a maze will not be changed while the program is running. So, you should construct this world in your application before transferring all vertices into the graphics pipeline. In doing so, it allows you to use the `glDrawArrays()` function only once to draw the whole scene.

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);
```

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 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 vec4 vColor;
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 / 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 texture lab:

```
#version 120

varying vec2 texCoord;

uniform sampler2D texture;
```

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

What to do?

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

Task 1: Dynamically Generate Mazes (10 Points)

Note that if you are going to generate a maze every time your program is executed, your maze should be a good maze:

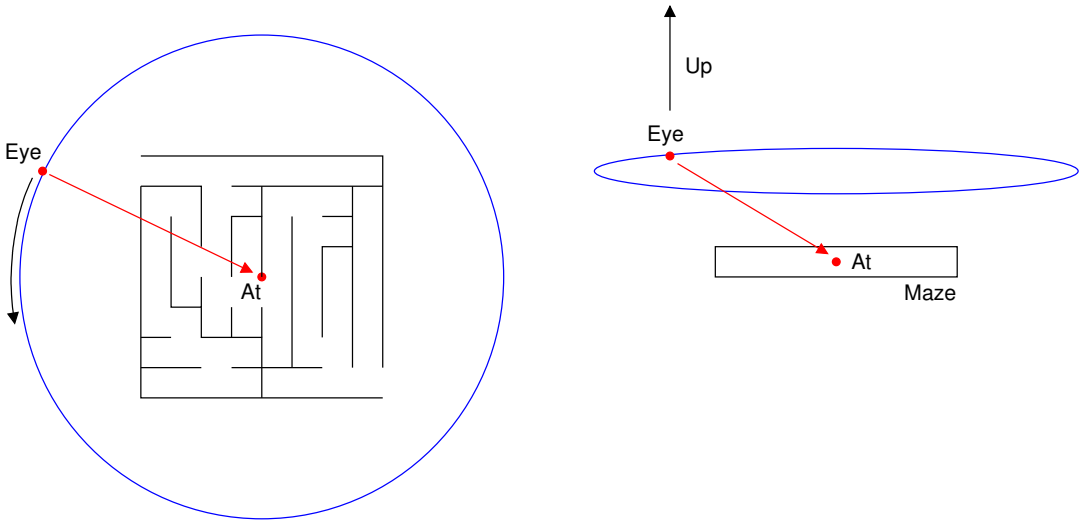
- There exists exactly one solution from one point in the maze to another
- There is no loop inside the maze

Task 2: Generate the World (30 Points)

Use the maze that you generate from the first part to generate a 3D world that consists of nothing but transformed cube. If you want, you use a more complicate objects. For example, use a cylinder as a pole. Your object must use texture(s) instead of colors.

Task 3: Flying Around the Maze (10 Points)

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 4: Flying Down (10 Points)

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 5: Solving the Maze (40 Points)

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.