OpenGL Random Maze Generator and Solution Finder

Daniel Meyer

Professor Tong Yu

CSE 420, Fall 2018

Objective

The objective for this project was to demonstrate knowledge of OpenGL and create a demonstration. My demonstration for this was the creation of an algorithm that randomly generates a maze, fill it with barriers and find the quickest solution from the entrance to the exit.

Abstract

The package for the maze includes two primary files. The first of these being Node.h which contains the data for each grid location in the system. The second file being maze.cpp contains all information for assembling the maze as then displaying a 3D representation of the maze. The maze itself is comprised of an array of std::list. To assemble the maze a series of function calls are made to assemble the maze after which the maze is then translated into a 3D representation. OpenGL and GLUT are used to create the 3D representation of the maze. GLUT objects are what the maze objects themselves are comprised of including: glutSolidCube() for the walls, glutSolidCone() for barriers, glutSolidSphere() for the exit, and glutSolidIcosahedron() for the solution path. OpenGL Quad primitives are used to assemble the floor and ceiling of the maze and OpenGL Line primitives are used to assemble the grid. Next, I utilized the Phong Lighting Model provided by OpenGL to add depth to the scene with diffuse and specular light. Finally, I used matrix transformation along with GLUT to animate the solution path as well as allow users to rotate the maze itself.

Introduction

The code used for the maze algorithm can be traced back to an algorithm based in Java that was used to develop a similar maze, however at the time it was not intended to be used with OpenGL or any form of 3D graphics. The code originally was a series of print commands to the console. Upon successful translation from Java to C++ I began focusing on converting the program from its console-based origins to an interactive 3D scene. The process of building an interactive 3D scene in OpenGL introduced me to a wide variety of new concepts I had never thought of prior. One of these concepts was using matrix multiplication in computer graphics to provide rotations, translations, and scaling to the maze itself as well as objects within the maze. Another significant learning experience this project provided was working within SDL to provide animation for the solution path within the maze.

Package

The ListMaze_3D_Final project is divided up between two files: Node.h and maze.cpp. Node.h comprises each grid space through the maze and is what each std::list is comprised of. The main source file, maze.cpp, contains all information for the maze creation algorithm as well as the code for creating an interactive 3D maze using the information generated by the algorithm.

The information stored within each Node includes as follows: an integer for if the Node is full, and integer for the index in the std::list of the Node, a Boolean value for if the Node has been visited during solution traversal, and three more Boolean values for the type of object filling the grid space. Each Node also includes a default constructor for creating an empty grid space with no object type. Lastly, each Node has a series of functions for assigning an object type (wall, barrier, exit, or player) to the grid space.1

The maze.cpp functions can be divided into two categories: the algorithm and the graphics and interaction functions. The algorithm portion of the code is the conversion of the original Java functions to C++ and follow a simple path from the buildMaze() function. The maze algorithm is based on an adjacency matrix that is used to create a grid that is then filled to create the maze. Each function is called by buildMaze() in a specific order to fill the adjacency matrix. At this point the OpenGL functions are called to begin drawing the maze. All of the functions related to creating the scene begin with "draw" for organization. The process of drawing each objects begins with traversing through each list in the array. As each node is visited, they are checked for the type of object (empty, wall, exit, barrier, player) and then an object is drawn accordingly. An example of this would be a barrier which begins with checking for isBarrier() == true then if true then the following occurs: the current matrix is saved, the color red is set, then the translation matrix is added (x-value = array index + provided offset, y-value = Node index + offset, z-value = 0 – offset), followed by a glutSolidCone() to represent the barrier, and finally the matrix is popped. This process is used for each of the maze objects with the difference being the shape drawn (blue rectangle = wall, red cone = barrier, purple ovoid = exit, green icosahedron = player).

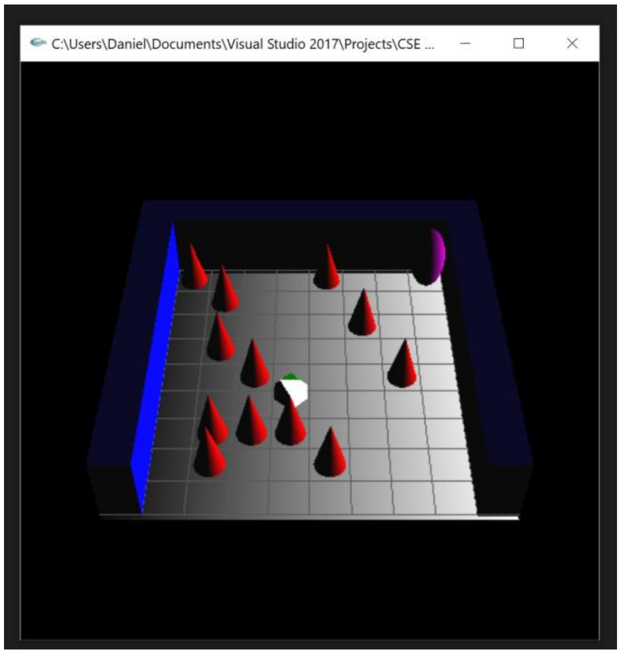
The maze.cpp file also utilizes OpenGL specific initializer functions necessary for drawing including display() and init(). Within init() GLfloat arrays are created to store information regarding the point light being used to illuminate the scene. glClearColor() is used to set the background to solid black, the glShadeModel is set to smooth to enable Phong shading followed by enabling GL_DEPTH_TEST for the lighting. Next the light source is added with proper diffuse and specular materials and then the light and lighting are enabled. The next block of code within init() sets up the camera by first setting the matrix mode to projection followed by using gluPersepctive to set the viewing volume. The last segment of init() is to call buildMaze() to generate the maze and fill the adjacency matrix. In display() the color and depth buffers are cleared followed by using gluLookAt() to position and angle the camera correctly. Next three glRotations are called with 3 different angle variables that are used to allow the user to rotate the camera using keyboard commands. Next a translation is made to center the maze on screen. Finally each draw function is called in order to fill the buffer and then glFlush() is called at the end to display the maze.

The remaining functions in maze.cpp are GLUT functions reshaping the viewport when the window size is modified and keyboard input commands for user interaction. There is also the inclusion timerHandle() and visHandle(), which are both used for animating the solution path. timerHandle is set to a 1000msec or 1 second delay between position changes, however there is an issue where any keyboard inputs will accelerate the animation due to glutPostRedisplay() being used to update the animation as well as for updating the rotation of the maze. I had made several attempts to correct this, however I ran out of time before the project was due for submission.

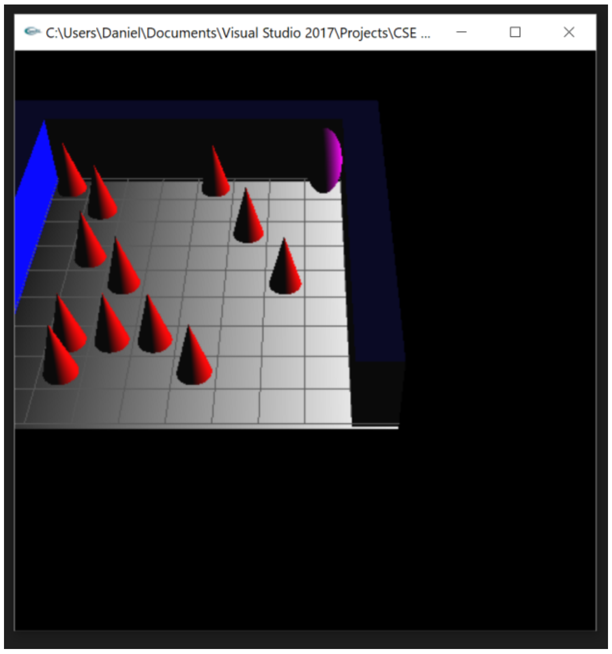
Interface

User interaction is handled through a series of keyboard inputs as follows:

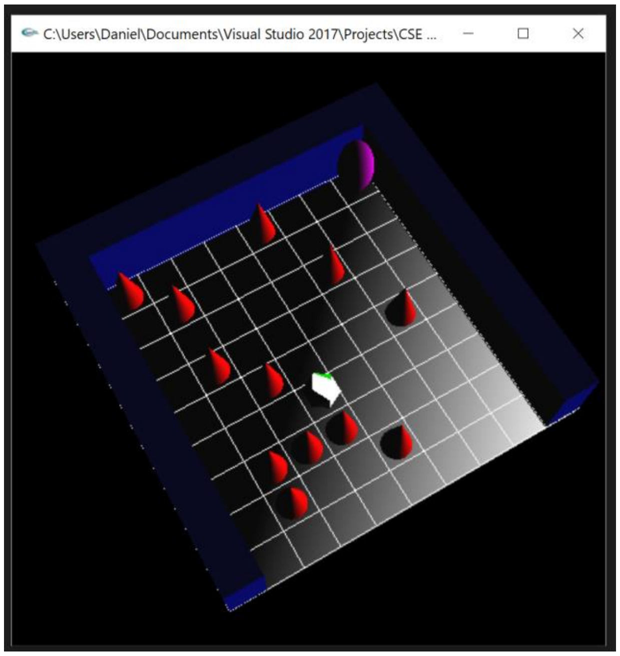
- 'up arrow' = rotate up (counter-clockwise about the x-axis)
- 'down arrow' = rotate down (clockwise about the x-axis)
- 'left arrow' = rotate left (clockwise about the z-axis)
- 'right arrow' = rotate right (counter-clockwise about the z-axis)
- 'w' = translate north (+ y-axis)
- 'a' = translate west (- x-axis)
- 's' = translate south (- y-axis)
- 'd' = translate east (+ x-axis)
- 'e' = translate up (+ z-axis)
- 'q' = translate down (- z-axis)

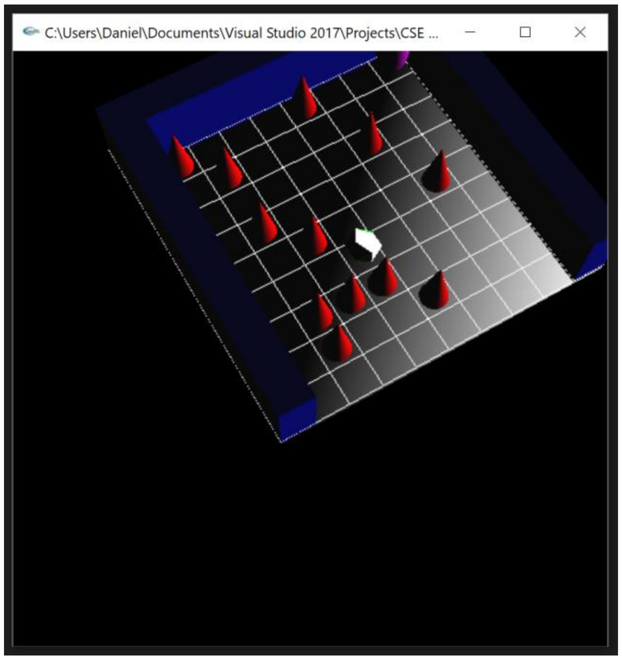

C:\Users\Daniel\Documents\Visual Studio 2017\Projects\CSE ...

Default View


C:\Users\Daniel\Documents\Visual Studio 2017\Projects\CSE ...

Translated left and up


C:\Users\Daniel\Documents\Visual Studio 2017\Projects\CSE ...


C:\Users\Daniel\Documents\Visual Studio 2017\Projects\CSE ...

Rotated right and down

Rotation with translation

Conclusion

This project let me experiment with the different aspects of OpenGL and GLUT. This project helped me further my understanding of the concepts of matrix multiplication in computer graphics as well as the concept behind generating 3D objects using OpenGL. Some areas of this project can be improved upon. One such are would be the animation for the solution path as it currently updates with a set interval, but also updates whenever translation or rotation is performed. This is due to both actions call glutPostRedisplay() and I have a possible solution using SDL, however due to time constraints was unable to implement and debug this feature. Another area of improvement could have been the lighting of the scene. Currently the icosahedron representing the solution path is blown out with white lighting and is hard to see the object's green color. The lighting on both the floor and ceiling could also be improved. This could be done by adding more polygons and fixing the light source's position. Looking to future work on this project I would like to implement better animation as well as the ability for users to solve the maze themselves. This would involve adding user input commands, a more complex maze, and collisions for the barriers. Overall, the project was an extremely beneficial experience.

References

Khronos Group. (n.d.). The Industry's Foundation for High Performance Graphics. Retrieved from https://www.opengl.org/

Khronos Group. (n.d.). The Industry's Foundation for High Performance Graphics. Retrieved from https://www.opengl.org/resources/libraries/glut/

Yu, T. L., Ph. D. (n.d.). CSE 420 Computer Graphics. Retrieved from http://cse.csusb.edu/tongyu/courses/cs420/index.php