Generating and Displaying Mazes in Two and Three Dimensions

Robert Kaufman
rbkaufm2

December 2015

Abstract

Mazes are a sort of puzzle in which the objective is navigate some space from a starting point to an ending location. The challenge arises from the layout of the seemingly random walls which restrict what paths may be taken. The most common examples of mazes are two dimensional but in this project, part of the goal is to extrapolate the concept of a maze into three dimensions. The second goal of this project is to experiment with different algorithms to automatically generate the maze with each providing slightly different visual styles and solution complexity.

1 Introduction

While mazes may be designed by hand, it is faster and easier to do so with a computer program, especially if the maze goes beyond 2D. This project will be showing how different algorithms develop a maze each with their own pattern and difficulty as well as how they translate into higher dimensions. Since the majority of the algorithms simply have different methods of selecting a random cell to open to, the translation of each of the four selected algorithms to the higher dimensions will not be too challenging. Another goal is for it to also provide an effective user interface and display system for a user to try and attempt the 2D and 3D generated mazes. Given the nature of mazes and the importance to see the large picture, this task is the more difficult of the two components of this project. With 3D, it becomes difficult to visualize the whole maze at once in order to try and see the one correct path to the end.

2 Maze Generating and Solving

In general, a maze can be considered a spanning tree with each grid location representing a node and the lack of a wall between each grid location as an edge. A spanning tree is a type of graph with n nodes and n-1 edges such that there is only one path from any node to another and every vertex is connected. This property is important as it means that a start and end location may be placed anywhere on the maze and there will be guaranteed to be just one true solution. An m-ary spanning tree is a spanning tree with each vertex having at most m edges connected to it. In context to this project, a 2D maze is a 4-ary spanning tree and a 3D maze is a 6-ary spanning tree. Though the implementation of a 4D maze was scratched, one could be represented by an 8-ary spanning tree. For any discussion of generating a maze in the rest of this paper, it will be in the context of creating a spanning tree. Whenever one vertex is said to be connected to another, it means that the wall is removed between one grid space to the other one represented by the other vertex.

To accomplish generating the mazes, four algorithms were coded. Each algorithm works similarly by focusing on a grid location or vertex. For the 2D maze, the initial formation would be a grid of unconnected vertices with each of the central ones having a neighbor to the north, south, east, and west, hence a 4-ary tree would be formed after generation. For the 3D maze, the initial formation is a box of vertices with each internal vertex having a neighbor to the north, south, east, west, above, and below, hence once generated, a 6-ary tree would be formed.

The *recursive backtrack algorithm, Prims algorithm, and the growing tree algorithm*, each start with all walls closed, or in regards to a tree, a fully disconnected graph of just vertices and no edges. In theory, the *recursive division* works with a fully connected graph (no walls present) and disconnects vertices until a spanning tree is formed but the implementation for the maze applications starts with a fully disconnected tree like the other three algorithms and adds edges until a spanning tree is formed. The function and implementation of each algorithm is discussed in more depth below.

2.1 Recursive Backtrack[2]

The first method implemented and one of the more pleasing ones is the recursive backtrack algorithm. This function works by first selecting a random vertex as the starting location. It then will randomly connect an edge to one of the four (if 2D) or six (if 3D) surrounding vertices (open the wall to one of the four surround grid spaces). Then, starting from the newly connected vertex, it will again randomly connect to another surrounding vertex such that it does not form a closed circuit. In practice, this means that it will randomly connect to any surrounding vertices not yet connected by an edge. It will continue this process by randomly connecting from the most recently added vertex until an edge can no longer be added. At this point, the algorithm will backtrack along the its path by going back to its parent (hence the name) and try to add an edge to that vertex. If it can, it will randomly make a new path from that one like before and if not, it will backtrack its parent and continue this pattern until every vertex has been visited. This is implemented recursively.

As mentioned previously, this is one of the most visual pleasing algorithms that also lends to mazes with longer solutions. The pattern created is very random looking and solutions tend to wind around for a longer distance than the other methods.

2.2 Recursive Division[2]

Like the other algorithms, this one starts with a fully disconnected graph comprised of only vertices and no edges. Consider the vertices for the 2D maze to be placed in the xy-plane and the vertices for 3D maze to be placed in a cube that is located fully in the first octant of the xyz-space.

For the 2D maze, the algorithm works by selecting randomly one of the x and y axis. It will then randomly select a line parallel to the selected axis located within the range of the graph. Each vertex is considered to be placed at an integer location for both the x and y position. This randomly selected line is selected in range such that its x (if parallel to the y-axis) or y (parallel to the x-axis) position is an integer increment plus 0.5 so that it will never intersect a vertex but instead be in-between two sets of vertices. At random, it will select one pair of vertices to connect with an edge such that they are on either side of the line and neighbor each other. The vertices on one side of this line are then considered a new graph and the vertices on the other side of the line considered a different graph. Each of these subgraphs undergoes the same process recursively until there are no more locations to place a line.

For the 3D maze, the algorithm works by selecting randomly one of the x, y and z axis. It will then randomly select a plane whose normal is parallel to the selected axis located within the range of the graph. Each vertex is considered to be placed at an integer location for each the x, y, and z positions. This randomly selected plane is selected in range such that its x (if the x-axis is selected), y (if the y-axis is selected), or z (if the z-axis is selected) position is an integer increment plus 0.5 so that it will never intersect a vertex but instead be in-between two sets of vertices. At random, it will select and connect one pair of vertices such that they are on either side of the plane and neighbor each other. The vertices on one side of this plane are then considered a new graph and the vertices on the other side of the line considered a different graph. Each of these subgraphs undergoes the same process recursively until there are no more locations to place a line.

As can be noticed, though both the 2D and 3D methods for this algorithm are similar, they still have more differences that have to be accounted than the other algorithms. The pattern of the recursive division is likely the least pleasing as it tends to create bland and simple mazes with large swathes of parallel lines with random gaps connecting them. The solutions are generally very straight-forward and short because of this.

2.3 Prim's Algorithm[2]

Prims Algorithm is more recognizable as an algorithm for finding the minimum spanning tree of a graph with weighted edges, but since the maze is considered a graph with each edge having equal weight, it will simply generate random paths. Because each edge is considered to have equal weight, the algorithm was applied to focus on the vertices instead of the edges.

This programs application of Prims algorithm starts by selecting a random vertex much like the recursive backtrack algorithm. It will then add that vertex to a list of visited vertices. A random vertex will be selected from the list and since only the first one is in it, that vertex will be selected. A random neighboring vertex from the selected one will then be connected by an edge if it has not been connected yet. If that edge is connected, it is added to the list of visited vertices. This process of selecting a random vertex and connecting one of its random, unconnected neighbors continues until all vertices are in the tree.

Prims algorithm also provides a pleasing random pattern, but there are many more vertices that are have the maximum number of edges or close to the maximum number of edges; for 2D mazes, there are comparatively more intersections of three or four paths and for 3D mazes there are more intersections of five or six paths. This means, that will there is more random paths, the solution is also relatively simply and straight forward.

2.4 The Growing Tree Algorithm[1]

The last and most diverse generation algorithm implemented is the growing tree algorithm. The unique part of this algorithm is that it can be tuned to create different variation of mazes depending on certain parameters.

It starts out just like the recursive backtrack algorithm and depending on the inputted parameters, it may behave completely like it. First a random vertex is selected and a random one of its neighboring vertices is connected to it. It will continue to connect a random, unconnected vertex until it is no longer possible, adding each new vertex to a list of vertices. Once a vertex can no longer be connected to any more vertices, it is permanently removed from the list. This is the point in which the algorithm differs and the input can alter the maze. The algorithm will use some method to select the next vertex from its list of visited vertices to continue making a path from. The algorithm terminates once the list of vertices is empty, signifying that every vertex has been connected.

For this application, the user is presented with the option to assign weights to the selection options used for choosing the next element to expand the path from. The three options are the newest element in the list, the oldest element in the list or a random element in the list. The program will then randomly choose one of those based off of the weights assigned and start at that vertex to continue expanding the graph once a dead-end is encountered.

The new option will make the algorithm behave just like the recursive backtrack as it will pick the newest vertex added to the list which will always be the parent vertex of the one that it ended on. Thus the pattern looks the same as the recursive backtrack with a pleasing, random pattern and long solutions.

The old option selects the vertex that has been in the list the longest. This means that at the first dead-end, the program will start the next path from the first vertex. This creates a very simple pattern with mostly straight lines emanating from the starting point and as such, creates a poor looking and very simple maze.

Finally, the random options acts as one would assume. The algorithm will randomly select the next vertex from its list to continue making a path from. Due to this random nature, the generation acts similarly to, though not exactly like, Prims algorithm. Thus, the generation of this option looks about the same.

2.5 Solution Algorithm

The final algorithm implemented is one to solve the maze and was done with a depth-first search which works similarly to the recursive backtrack algorithm. Starting from the start vertex, the algorithm proceeds to try and find the solution by creating a path down each of its neighbors until the end vertex has been reached. This method is recursive in which each neighbor visited also follows the same process. If a path down one neighbor reaches a dead-end, it backtracks to its parent and tries the next neighbor to repeat the process. The neighbors are selected in the same order every time; its goes in the order of up, down, left, and then right with the 3D solver continuing to above and below.

3 Code Structure

In order to simplify the process of creating, solving, and generating the mazes, the code was structured using a Cell object to represent each vertex or cell in the maze. This object held an array to represent the walls surrounding it, whether it had been visited yet or not, whether it was the start, whether it was the end, and whether it was part of the solution. The x, y and if the cell is part of the 3D maze, z positions were all stored as well. These cells were stored in a global 1D array called cells which represented the full maze. This made traversal of each cell and retrieval of a specific cells necessary data simpler.

4 Displaying the Mazes

The mazes were displayed using javascript as the base code and the 2D HTML5 canvas library set for the 2D maze and WebGL and ThreeJS for the 3D maze. This allows the applications to be portable on a website and operable on most computers.

4.1 2D Display

The process for displaying the 2D maze was relatively simple. First, the size of the canvas drawing space was retrieved in pixels and based on the number of grid elements in the maze and the pixel size of each grid cell was calculated. If it would have been smaller than the preset minimum size of 10 pixels, the grid cell size would be set the 10 pixels and only part of the maze would be drawn at a time. If this is case, the maze could then be translated and zoomed in or out of in order to view the full maze. A maximum and minimum zoom as well as a max and minimum x and y shift are then set as to allow the full maze to be viewed without the possibility going to extremes and losing it.

After this initial process, the maze is drawn in two main steps. First, each grid cell is drawn on the canvas as a filled square of side length equal to the minimum grid size times the level of zoom. If it is an empty cell, the color drawn is simply white. If a maze has been generated and a start and end are created, then the start cell will be drawn as a muted yellow square and the end cell will be drawn as a muted green square. In order to help display the solution, be it computer generated or user generated, cells are marked as a light blue, a muted red, or a darker blue. The light blue cell represents part of the current solution and the red cells represent each cell that has been visited in order to try and find the solution. The dark blue cell marks the current position of the user; it will not be displayed if at the start or end.

The second step of drawing the maze is drawing the walls. As each cell is an object with an array of walls, this is done by looping through every cell in the grid and drawing a line representing each wall that is present. The width of each line is 2 pixels and the length is one cell size. In order to maximize the number of cells that can be displayed, cell size was made to be a floating type variable. This means that the lines will try to draw over a fraction of pixel at times causing fuzzy edges as the rendering system tries to represent that with blending the pixels on either side of the edge.

4.2 3D Display

Displaying the 3D maze proved to be a much more arduous endeavor in both programming the display as well as determining the best way to display the maze such that a user can try and see and solve the completed maze. While 2D canvas could still be used to manually display 3D content, it becomes a more tedious task so instead, WebGL along with the ThreeJS library was utilized. With this, a ThreeJS scene, camera, and renderer had to be initialized at the start of the application. Along with this, variables controlling the maximum and minimum camera zoom were set based upon the maximum of the number of cells in the height, width, or depth. With the 3D scene initialized the 3D maze could then be displayed. There were two methods selected to display the maze that could be toggled between. One was the standard view which would show the full rectangular prism with lines marking the paths on the surface.

In order to display this maze, a ThreeJS geometry called geom is created and initiated with a list of vertices that represent the eight corners of each individual cell. Then, looping through each cell in the maze, a function called genGeometry would add a face for each wall of that cell to the geom and assign to them the default, white, opaque material. The faces were then assigned a color based off of the same color system used for the 2D maze. After geom was filled with the faces for each wall in each cell, a new ThreeJS mesh was created from it called grid. A wire frame called edges was also created from geom. This wireframe would display the maze paths on the outer surface. Finally, both the new mesh and wireframe were added to the scene and rendered.

The problem with this view mode was that it was impossible to see how the maze formed underneath the surface and as such, it is not an ideal mode to try and solve the maze. To combat this problem, a second method of display was implemented. This second method shows a single layer of the maze. The focused layer would be the one facing towards the camera. The Q and E buttons would then control that layer and allow it to shift up and down such that every layer of the maze could be viewed.

Similar to the normal display method, a ThreeJS geometry called geom is created and filled with the appropriately colored faces (to represent the start, end, current cell, visited cells, and solution cells just like the 2D maze) to represent each wall of the cells that are in the focused layer. The top-facing wall of each cell is not displayed in order to allow the user to see into that layer and at the paths that may be taken. The vertical wall faces in this mode are also set to a different basic material that is a darker color and translucent. This helps to provide contrast between the vertical walls and the floor-wall. In order to determine which wall of the cell were the vertical walls and which were the floor-walls, a variable kept track of the rotation and would change to represent the current face of the rectangular prism that faced towards the camera.

5 Controlling and Solving the Mazes

In order to make the mazes interactive and interesting, some control must be presented to an outside user. This is done through a combination of mouse clicks and movements, keyboard presses, and buttons present on the webpage containing the maze application.

5.1 Common Inputs

There is a set of inputs common to both the 2D and 3D mazes that generally concern the generation and solving of the mazes. The first of these options are buttons and text labels visible on the webpage below the maze applications. The first row of these buttons is the maze generation buttons labeled Recursive Backtrack, Recursive Division, Prims Algorithm, and Growing Tree respectively. A press of any of these buttons will generate a new maze by the specified method and redraw the maze. When pressed, the Growing Tree button will also update the weights of its parameters according to values in the text boxes in the second row.

The third row consists of two text boxes for the 2D maze and three for the 3D maze. These text boxes represent the number cells in the width, height, and depth respective to their label. Pressing the Enter button will reset and remake the maze with the new dimensions.

Finally, the fourth and last row of inputs is three buttons labels Solve, Clear Solve, and Reset Maze. As can be guessed, the Solve button runs the depth-first-search algorithm and displays the solution of the maze. The Clear Solve button will undo any user or computer generated solution and returns the current cell position back to the start cell. Reset Maze resets the whole maze back into a grid of all walls (a graph of with just vertices and no edges).

The next set of common controls is the keyboard controls. The R key, when pressed, performs the same function as the Reset Maze button and the F key performs the same task as the Clear Solve button. Aside from changing the maze, the controls used to move the current cell position in order for a user to try and solve the maze is W, A, S, and D for moving in the up, left, down, and right directions respectively. The current cell will only move in a direction if a wall is not blocking its path.

Finally, both the 2D and 3D mazes share a control to manipulate the viewing of the maze. Both applications can be zoomed in by pressing the combination of the shift key and left mouse button and translating the mouse pointer over the application. The 2D maze uses the absolute distance change determine how much to zoom in and the 3D application only uses the difference in the y positions of the mouse. This difference was made simply because each option felt more responsive and in control of their respective maze.

5.2 2D Controls

Due to the simpler nature, the 2D maze application does not have many specific controls. In fact, it only has one, the mouse control to translate the section of the maze in view. As discussed in the 2D Display section, if a maze is too large, part of will be truncated from view. By clicking the left mouse button and moving the mouse, the mouse will drag the maze with it to allow other sections to be viewed.

5.3 3D Controls

With an extra dimension and two display modes comes a greater number of controls, all of them handling how the maze is displayed. As discussed in the Display 3D section, there is a normal display mode and a layer display mode. The G key on the keyboard toggles this while the Q and E keys on the keyboard increase the layer level and decrease the layer level respectively. Another control added to help with viewing the 3D maze were the assignments of the 1-6 number keys to a face on the rectangular prism. Pressing one of those keys will instantly snap the view to that face.

Finally, a control to manually view different sides was added in the form of rotation. If the left mouse button is pressed and translated, the rectangular prism will try to rotate to follow. For example, if the mouse is dragged to the left, the maze will rotate in the negative direction in the along the y-axis. If the mouse is moved up, the maze will rotate positively over the x-axis. If the maze is rotated a half-turn upwards or downwards, the program will flip the horizontal rotation as well as solution movement as the viewed face is now upside-down.

6 Conclusion

Though the main goals of the revised project goals were accomplished, some aspects could have been addressed better. For example, the table of button controls could have been better integrated into the canvas applications and the 3D maze could have more well defined wall markings as well as a possible first-person solving display in order to make navigation easier. Extension to 4D is still an interesting concept and generating and solving the 4D mazes would be simple as it would just be creating 8-ary spanning trees. Yet, as evident with the struggles to display just the 3D maze well, displaying a 4D maze effectively would be immensely complicated if not impossible.

References

[1] Jamis Buck, http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

[2] https://en.wikipedia.org/wiki/Maze_generation_algorithm