3D Surface Pathfinding Algorithm

Anish Dhesikan

Professor Nik Brown

Algorithms & Data Summer I

June 29, 2015

**Abstract**

There are many methods of finding a path from one point to another in 2D space. One may believe that extending this problem to a three-dimensional surface may make the problem much trickier. However, extending this problem to the third dimension has proven to be very similar to the 2D pathfinding algorithms that are known today. After a couple attempts at finding an algorithm that accomplishes the goal and always terminates, I have produced one that is successful. The algorithm uses a bread-first-search technique to find the shortest path between two points along a tiled surface in 3D space. This can be utilized for a wide array of situations, including spider AI that can only travel along surfaces, as well as wiring inside walls of buildings.

**Introduction**

There are several methods of pathfinding in 2D space. One of the most common algorithms is the A\* (A-star) algorithm because it is known to run faster than Dijkstra’s or Breadth First Search (BFS). The algorithm uses a heuristic of predicted distance in 2D grid space to dynamically calculate the optimal / shortest path from one point to another (including diagonal movement along the grid). For the problem I wished to solve, I did not want diagonal movement. Thus, I began to think of alternate, faster solutions for pathfinding with simple movement along the grid. I began by thinking in 2D space, and found that the optimal algorithm would be the greedy algorithm, simply starting at a tile and selecting a neighbor (adjacent tile) that is closer to the end point until reaching the end tile. This would work in 2D space, but using 3D distance as the heuristic and using the same algorithm in 3D space did not always terminate because there are cases in which none of the adjacent tiles are closer in 3D space. Thus, I decided to use something like BFS instead. See the code section for the actual implementation of the algorithm.

**Code with Documentation**

First, I will provide some pseudocode to explain the code in a simpler way.

***Part 1. Connecting adjacent tiles:***

For each tile

Cast 12 rays to detect all possible neighbor cells in 3D space.

Create 4 pointers from this tile to each of its neighbors for accessibility.

***Part 2. BFS and Backtrace:***

Initialize an empty queue.

Add just the starting tile to the queue.

While the queue is not empty

Get the first tile in the queue.

Add each of its neighbors (if not already visited) to the queue and set each of the neighbors’ parent to this tile until the current tile is the goal tile.

Once the goal is reached, break out of the while loop.

Trace the path by starting at the end tile, adding its parent to the list of in-path-tiles, and continuing to add the parents until the start tile is reached.

Return the list of in-path-tiles.

Now, here is the code. Written in C# using Unity (Game Engine) framework. This script is attached to each tile in the scene.

***Part 1.***

// **Assigning neighbors** to each tile (cast 12 rays to find this tile’s neighbor tiles)

    void AssignNeighbors () {  
        Ray ray;  
        RaycastHit hit;  
  
        ArrayList directions = new ArrayList();  
        directions.Add (transform.up);  
        directions.Add (transform.right);  
        directions.Add (-transform.right);  
        directions.Add (-transform.up);  
  
        foreach (Vector3 dir in directions) {  
            ray = new Ray (transform.position + transform.forward / 2, dir);  
  
            if (Physics.Raycast(ray, out hit, 1f)) {  
                if (dir == transform.up) {  
                    this.tileUp = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == transform.right) {  
                    this.tileRight = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == -transform.right) {  
                    this.tileLeft = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == -transform.up) {  
                    this.tileDown = hit.collider.GetComponent<PathfindingTile>();  
                }  
            }  
  
            ray = new Ray (transform.position - transform.forward / 2, dir);  
              
            if (Physics.Raycast(ray, out hit, 1f)) {  
                if (dir == transform.up) {  
                    this.tileUp = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == transform.right) {  
                    this.tileRight = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == -transform.right) {  
                    this.tileLeft = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (dir == -transform.up) {  
                    this.tileDown = hit.collider.GetComponent<PathfindingTile>();  
                }  
            }  
        }  
  
        ArrayList origins = new ArrayList();  
  
        origins.Add (-transform.forward / 2 + transform.up);  
        origins.Add (-transform.forward / 2 + transform.right);  
        origins.Add (-transform.forward / 2 - transform.right);  
        origins.Add (-transform.forward / 2 - transform.up);  
  
        foreach (Vector3 orig in origins) {  
            ray = new Ray (transform.position + orig, transform.forward);  
              
            if (Physics.Raycast(ray, out hit, 1f)) {  
                if (orig == (Vector3) origins[0]) {  
                    this.tileUp = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (orig == (Vector3) origins[1]) {  
                    this.tileRight = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (orig == (Vector3) origins[2]) {  
                    this.tileLeft = hit.collider.GetComponent<PathfindingTile>();  
                }  
                if (orig == (Vector3) origins[3]) {  
                    this.tileDown = hit.collider.GetComponent<PathfindingTile>();  
                }  
            }  
        }  
    }

***Part 2.***

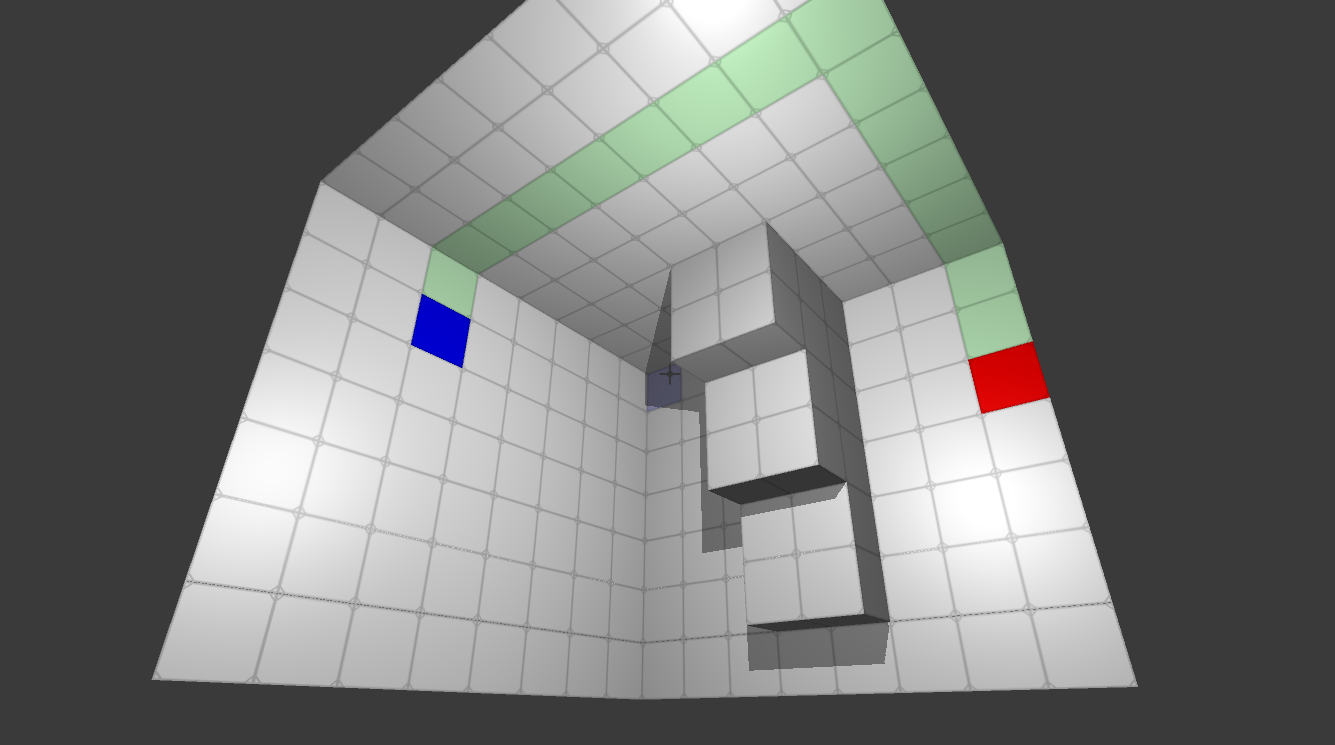
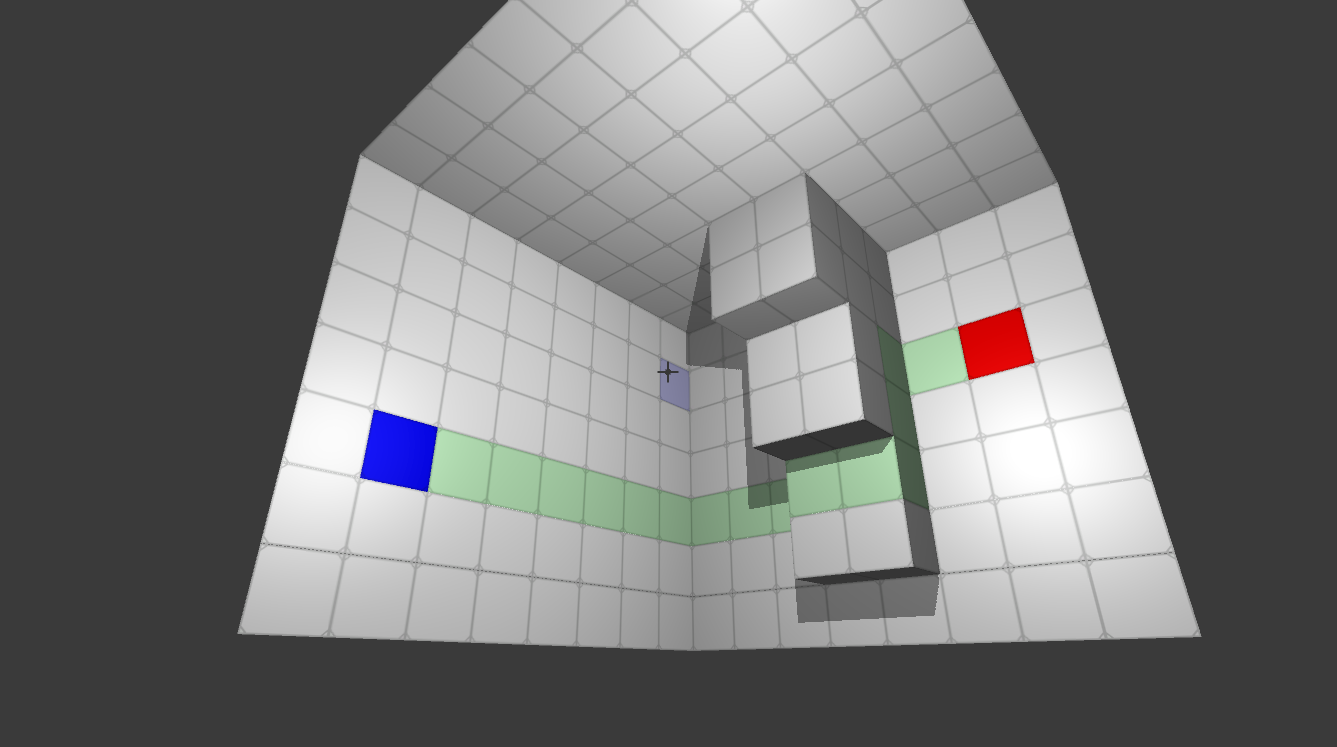
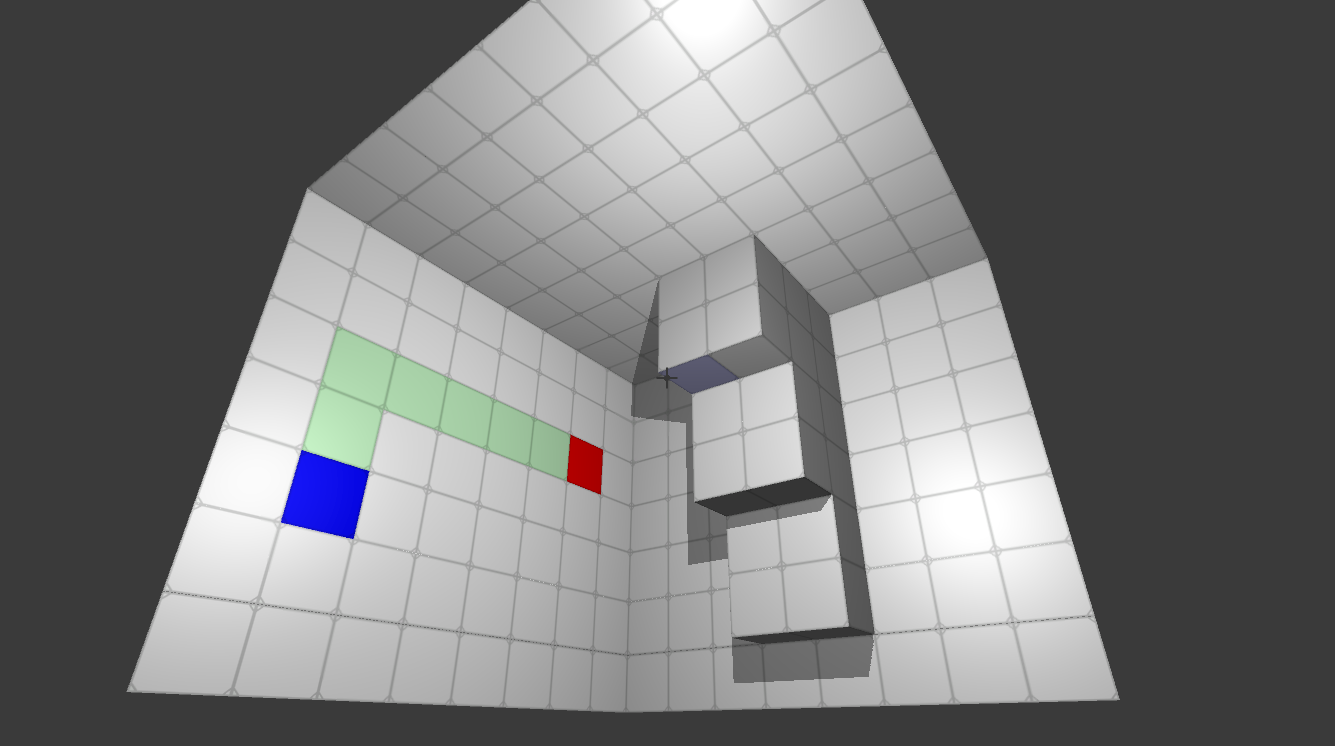
// **BFS and Backtrace.**

public void FindPath () {  
        inPathTiles.Clear ();  
  
        ArrayList visited = new ArrayList ();  
        visited.Add (this);  
        Queue queue = new Queue ();  
        queue.Enqueue (this);  
  
        *// BFS*  
        while (queue.Count > 0) {  
            PathfindingTile tile = (PathfindingTile) queue.Dequeue();  
  
            if (tile == endTile) {  
                break;  
            }  
  
            if (tile.tileUp && !visited.Contains (tile.tileUp)) {  
                visited.Add(tile.tileUp);   
                tile.tileUp.parentTile = tile;  
                queue.Enqueue(tile.tileUp);  
            }  
            if (tile.tileRight && !visited.Contains (tile.tileRight)) {  
                visited.Add(tile.tileRight);   
                tile.tileRight.parentTile = tile;  
                queue.Enqueue(tile.tileRight);  
            }  
            if (tile.tileLeft && !visited.Contains (tile.tileLeft)) {  
                visited.Add(tile.tileLeft);   
                tile.tileLeft.parentTile = tile;  
                queue.Enqueue(tile.tileLeft);  
            }  
            if (tile.tileDown && !visited.Contains (tile.tileDown)) {  
                visited.Add(tile.tileDown);   
                tile.tileDown.parentTile = tile;  
                queue.Enqueue(tile.tileDown);  
            }  
            Debug.Log(tile.name);  
            Debug.Log(queue.Count);  
        }  
  
        *// Backtrace the path*  
  
        PathfindingTile nextParent = endTile.parentTile;  
        while (nextParent != startTile) {  
            inPathTiles.Add(nextParent);  
            nextParent = nextParent.parentTile;  
        }

}

**Results**

The BFS algorithm (currently implemented) works to find one of the shortest paths between two tiles (there may be multiple shortest paths, but the algorithm is biased toward a certain direction). See the images below for the final outcome.



**Discussion**

The implemented BFS algorithm runs in O(V + E) time, which in my case is O(V+4V) = O(5V) because each tile has 4 edges so the number of edges E is the same as 4V. This algorithm is relatively fast, running in polynomial time, meaning it will scale well to larger numbers of vertices / tiles. The A\* pathfinding algorithm in 2D space runs in O(logV) time with the perfect heuristic. Finding the perfect heuristic in 3D space may be a bit more difficult, but the A\* algorithm would be faster than the current BFS algorithm given the perfect heuristic. More research will need to be done to find the best heuristic in 3D space in order to implement the A\* method in 3D space. Both methods will be easily extensible to include diagonal movement.

**References**

<http://www.policyalmanac.org/games/aStarTutorial.htm>

[https://en.wikipedia.org/wiki/A\*\_search\_algorithm#Complexity](https://en.wikipedia.org/wiki/A*_search_algorithm#Complexity)

<https://en.wikipedia.org/wiki/Breadth-first_search#Pseudocode>

<https://www.ics.uci.edu/~eppstein/161/960215.html>

<http://www.eecs.yorku.ca/course_archive/2006-07/W/2011/Notes/BFS_part2.pdf>