# Professor Please

15-466 P2: Pathfinding Documentation

Anna Etzel (aetzel), Ivan Wang (icw), Jun Huo (jhuo)

## Overview

In this project, ten student characters with the *Pathfinding.cs* behavior track and follow the professor (player character) in real time via A* graph search.

As in P1, every character prefab GameObject contains *Movement.cs*, the master class which takes other behavior scripts and scales them appropriately. The resultant vector is applied to ThirdPersonCharacter.Move, a script from Unity standard assets that applies movement to the character.

## Testing

This project has one relevant scene, a maze of corridors, ramps, and bridges (Scenes/Pathfinding.unity). The Professor character is player-controlled by the arrow keys and can jump gaps with the Space bar. All other characters are students who track the professor using pathfinding.

Included in the Demo/ folder are two videos of the scene. We provide one with Debug.Gizmos on (drawing the nodes and paths each character follows) and one without.

## Pathfinding

### Graph Construction / 3D Planning

The code for graph construction can be found in *Pathfinding/Graph.c*s. Essentially, we create a 3D grid where the x and z coordinates (the floor plane) are spaced by a user-defined distance *dist*, with a separate spacing for the y coordinates (height from floor) to make ramps easier to handle. We also take user-defined start and end coordinates, and create a grid that starts at the specified start coordinates and just exceeds the end coordinates in each direction, if necessary. Then, for every valid point in our map, we raycast to adjacent points, which are those north, south, east, and west on the current layer, the layer above, and the layer below. If that raycast does not hit the walls or floors, we consider those points a potential path and put them in a neighbor adjacency list for our later A* search.

To determine if a point is valid or not, we do a raycast downwards to see if the point is within the distance *dist* from the floor. We also do a “ledge” check to ensure that the point is not so close to the edge that characters doing pathfinding will fall off of them. This involves casting rays downwards from points a short distance (defined as *checkwidth*) away from the original point in each cardinal direction. At least one opposing pair must be the same distance away from the ground as the original raycast, as all ramps are facing in a cardinal direction. The other pair must both be within a range determined by the maximum slope a ramp can be (approximately 0.5 in our map). If these tests pass, the point is marked valid and can be a neighbor/have neighbors in the neighbor array.

### A* Search

After the graph is constructed, we perform A* to find the best path (see *Pathfinding.cs*). The algorithm runs the shortest path algorithm while keeping track of a visited set of nodes, a priority queue “frontier” of nodes to explore, and a map of nodes to parents (the predecessor each node was visited from). While the frontier is non-empty, the closest node is popped off and unvisited neighbors are added to the frontier with a priority of their distance plus a heuristic value. We use a basic heuristic calculated based off Euclidean distance to the target.

When A* has found the target, the algorithm terminates and the path is reconstructed by following the parent map backwards from the target. The path is slightly modified to improve realism of motion. First, any nodes behind the character are removed. Next, we delete intermediate nodes (shortcutting) if there is a straighter path that doesn’t use them; this smooths out zigzags that naturally form from a grid-based path. The resultant path is saved to the character so they can follow it in subsequent frames. We compute the path every few frames for performance reasons.

## Collision Avoidance

In addition to pathfinding, we utilize a collision avoidance script (*Collision.cs*). If a character sees another tagged with “Collide” in the middle of pathfinding, they will turn to avoid them, straying off the path briefly. Once the character is no longer in the line of sight, they will recompute a path and continue the path following.