Project 1: Path Planning

Due Date: Friday, February 24th, 11:59 PM

1 Introduction

I hate getting lost. So do videogame characters. Your videogame characters thus far have been sad and lost in their level, wandering semiaimlessly in an attempt to fulfill their basic needs and inner programming. Help them.

In this assignment, you must create characters that are able to get from point A to point B without falling into point C, which is a large hole, fire, etc. In order to do this, you will be writing a pathplanning algorithm. Your character must be able to navigate a series of waypoints, specified in your map-file. The endgoal for this assignment is an NPC capable of "capturing" a character you control.

2 Dojo

In the previous assignment you did not have to spend much time interacting with Dojo itself; however, in this assignment you will have to interact with the map geometry loaded into Dojo. We have provided a simple class called a Waypoint that is specified in the map file. Your job is to create a "line-of-sight" connectivity graph of the waypoints when you load a map. Basically you have to determine if you have a straight line path from one way point to another. We have provided you with the World::rayCast method to do this (you probably want to cast multiple rays from the perimeter and the interior of your NPC to make sure nothing is blocking its path; and remember to make sure that your NPC will not fall through a hole, so make sure you check the ground along a given path). You will then search this connectivity graph in game to find the best path for your NPC to use. You will also be responsible for getting your NPC to and from waypoints, i.e. once you navigate to the waypoint nearest the player you may have to leave the waypoints to "capture" the player.

There is also example code on how to use Quake 2 models as your characters. This is in no way required, but you may be tired of just using a plain ball so we thought we'd include it.

2.1 A*

This is pronounced ay-star. It is one of the simpler pathplanning algorithms, and is easily one of the most well-known and oft-used. It's a variation of Dijkstra's search algorithm where you use a combination of a heuristic and cost. The cost in this case will be physical distance. It is guaranteed to give an optimal solution given a good heuristic, but it is not guaranteed to give the solution in the fastest amount of time.

Here's a description of A* as taken from Gamasutra
(`http://www.gamasutra.com/features/20010314/pinter_01.htm`)

> The A* algorithm is a venerable technique which was originally applied to various mathematical problems and was adapted to pathfinding during the early years of artificial intelligence research. The basic algorithm, when applied to a pathfinding problem, is as follows: Start at the initial waypoint (node) and place it on the Open list, along with its estimated cost to the destination, which is determined by a heuristic. The heuristic is often just the geometric distance between two nodes. Then perform the following loop while the Open list is nonempty:
>
> * Pop the node off the Open list that has the lowest estimated cost to the destination.
> * If the node is the destination, we've successfully finished (quit).
> * Examine the node's neighboring nodes.
> * For each of the nodes which are not blocked, calculate the estimated cost to the goal of the path that goes through that node. (This is the actual cost to reach that node from the origin, plus the heuristic cost to the destination.)
> * Push all those nonblocked surrounding nodes onto the Open list, and repeat loop.

You don't have to use A* to search your connectivity graph if you don't want to. We're just recommending it.

2.2 Waypoints

To specify a waypoint in your map, use the path_corner entity in GtkRadiant. We're using line-of-sight waypoints for pathfinding, so it's very important when creating test maps that each waypoint can "see" at least one other waypoint. This makes proper map design very important.

3 Requirements

You should turn in a working bot that can "capture" (i.e. chase/follow) a human controlled character. We provide you with a simple example map to use for testing. But you should definitely build, and turn in, your own maps to test more complicated cases and show off your AI.

4 Getting Started

The first thing you should do is create the connectivity graph. It is important to make sure that this is correct before you start writing your search algorithm because it will make the search algortihm easier to debug. Also, this assignment is considerable more work than the previous assignment, so make sure you get an early start.

5 Handing In

5.1 What to Hand In

Make sure that you hand in all of your code as well any maps and textures. Along with this, you must handin a README in which you explain your design choices in creating your map as well as the changes you made to Dojo.

5.2 Handin Script

IMPORTANT: Make sure to clean your solution (Build>Clean Soluction in Visual Studio) before handing in.

To handin your project, run /u/course/cs196-2/bin/cs196-2_handin asgn1 in your project's the base directory.

6 Extra Credit

Line-of-sight waypoint algorithms are not great. As you may notice, your characters' movement is largely dependent on the map file's waypoints: if two waypoints should be traverseable but the character thinks they're not, that generally means that more waypoints are needed as a guide. So, do real pathplanning without waypoints. That would be cool. See the following article on 3D pathplanning:

`http://www.gamasutra.com/features/20020405/smith_01.htm`

If you're interested in this let us know and we can give you code to easily access the map geometry itself.