LICH

Artificial Intelligence Toolbox

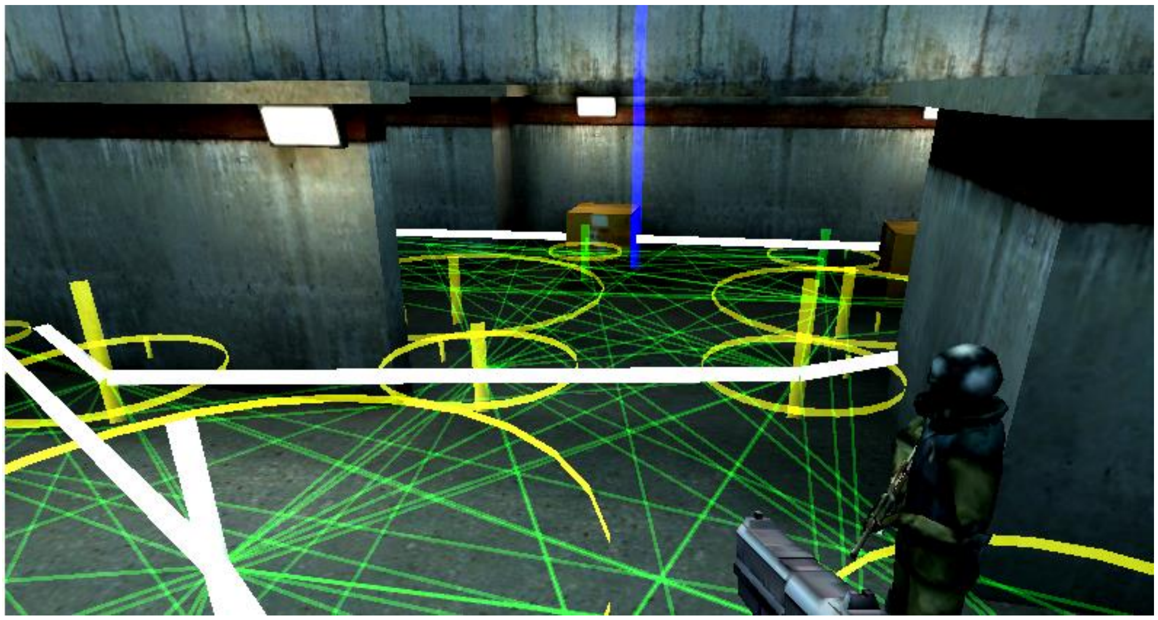
Navigation Point System & Path Finding
Technical Reference Document
[8.7.2001]

Contents

Introduction

Navigation and path finding are arguably the most visible, complicated, and unpredictable aspects of game Artificial Intelligence. For that reason, LICH incorporates a heavily preprocessed system of navigation points for the game agents to follow. These points are an agent's only real knowledge about the static architecture of the world in which it lives. Obviously, it is critical that these Navigation Points are placed to give the agent the most useful, abundant, and readily available information possible.

In many ways, the quality of LICH AI is in the hands of the person placing Nav Points.



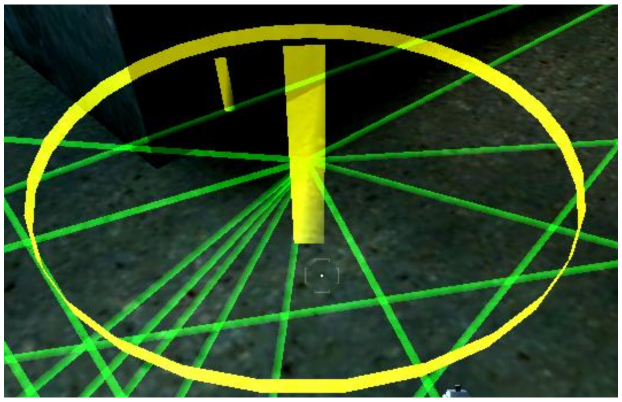
This document is intended to provide an in-depth discussion on the Navigation System and Path Finding, and does not contain any details on the actual execution of path *following*. For information on Architecture and Entity Avoidance, Flocking, Seeking, and other path following related techniques that LICH provides, please review the document on the Motor Controller Module.

Anatomy Of A Navigation Point

Every Nav Point contains a few fundamental elements:

ORIGIN:
When read in from a file or created on the fly by the player, the origin is where everything for a nav point begins. The point will always try to trace down to the ground from its start point to ensure that it is exactly 25 units off the ground. The point's coordinates are also snapped to integers to simplify things.

The origin ground trace will begin 30 units up from the initial position and trace down 150 units from there (with the exception of random terrain height map generated points). That is the extent of a point's ability to adjust to changes in architecture. If the floor changes any more than that and the point will most likely be removed.



ROOF, VISIBILITY, & MATERIAL:
When a point does hit a ground surface, it stores the material type of that surface, the light visibility at the origin as a number between [0.0, 1.0], and the height of the roof from a trace that goes straight up.

WALL:
After finding the origin, every nav point will cast out 36 rays of 100 units in 10-degree increments searching for the closest obstructing wall architecture. If it hits anything, the strike point is stored in as the "Wall", but is sometimes referred to in older code as the "Intersection". The wall point is pulled back from the real wall toward the origin by a few units because agents often want to get right up to the wall and cannot get exactly there due to bounding boxes.

If no wall is found within a 100-unit radius, the wall point is set to the same coordinates as the origin.

RADIUS:
By simply taking the distance from the wall point to the origin point we calculate a radius for the nav. Radius is a fairly important part of the path fining system, because it culls out a lot of distance checking and tracing when an agent is finding its position within the grid.

The more ground area is covered by nav point radii: the less tracing LICH needs to do.

COVER TYPE:
If the nav successfully discovered a wall, it will attempt to see what kind of architecture is there by tracing in that direction for several height levels. The first trace that goes farther than the radius is marked as following:

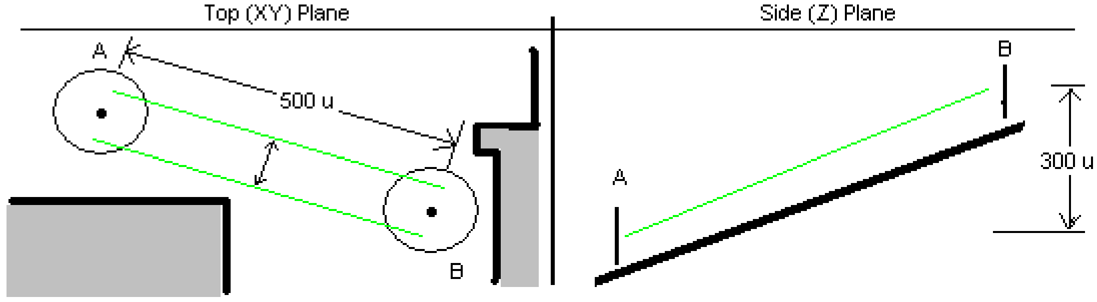
Type	Comments	Height	Color
Open	No cover, did not hit a wall within 100 unit radius	-	White
Prone	To hide from fire, must go prone, but can shoot from crouched	0 - 20	Red
Crouch	To take cover must crouch, but can stand and shoot	20 - 50	Green
Stand	Can hide by standing, can only attack with grenade	50 - 80	Blue
Wall	Wall extends all the way to roof	> 80	Yellow

Connecting Nav Points

Points alone do not provide enough information to game agents to be useful. The greatest power of the navigation grid is in the paths that it designates. The rules for when and how two given nav points (A and B) connect to each other and build a "Nav Bridge" follow:

Basic Standard (Walking/Running) Bridge

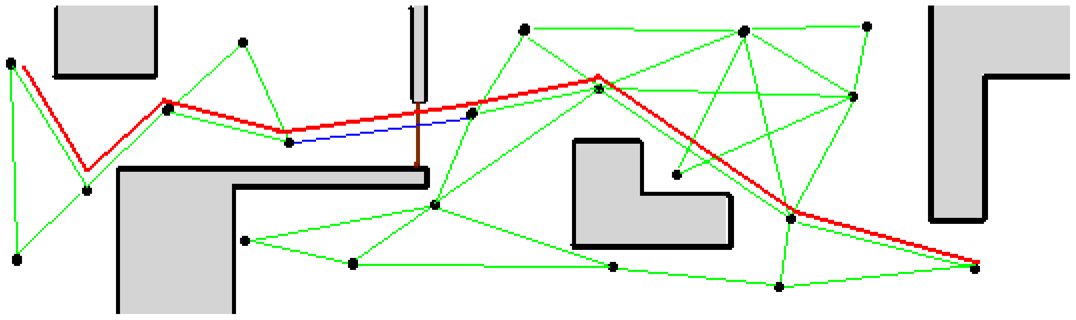
- XY Plane Distance < 500 u
- Z Height Distance < 300 u
- A Bounding Box Dimensions [26, 26, 20] Must Be Able To Move From A to B (green lines)
- No "Holes" (See definition of a Hole) Between A and B



Path Finding With A*

A* is what is known as a heuristic hill climbing algorithm. When called upon to find a path from A to B, it will "explore" the nav point grid in the direction of the end point, but at the same time keeping track of the distance it has traveled from the start. Other algorithms for path finding simply explore in semi random directions or do not care about distance traveled.

- *A* WILL find the shortest path if one exists.*

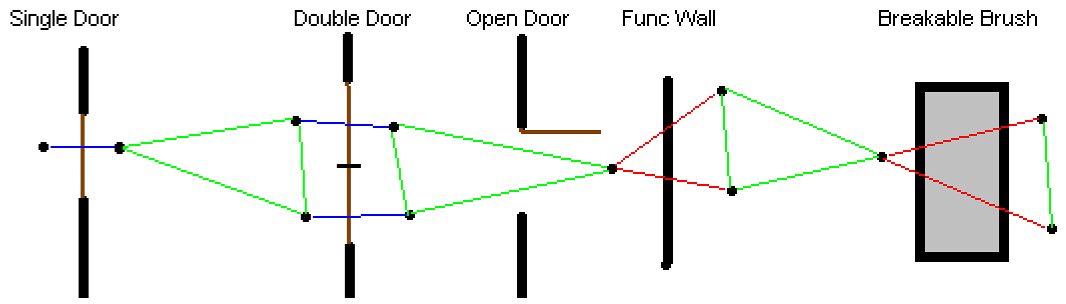


- The worst case for A* path finding would be when the objective point is very close to the start point, but the path to get there goes far out of the way and has many similar possibilities. (Imagine a spoke wheel where both the start and end are at the center, but in order to get from one to the other you need to go out to the rim).
- Several modifications have been made in our version of A* to handle different cases, like locked doors, one way bridges, and points that are blocked by an entity.

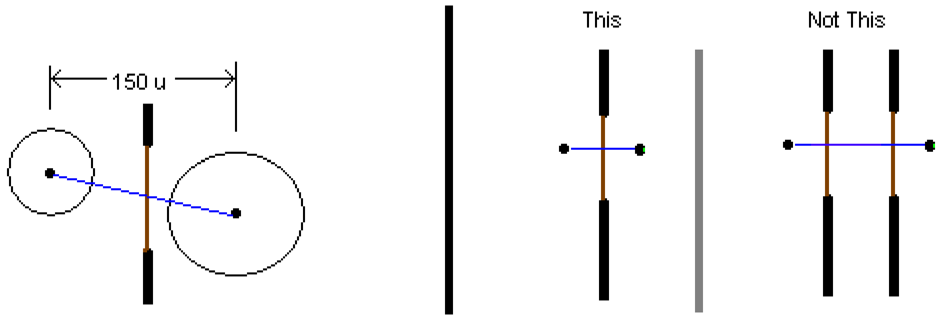
Bridge States And Doors

The existence of doors creates some unusual problems for path finding, as doors can have different states and these states determine the availability of the bridge. Therefore each bridge has 3 corresponding states:

OPEN – This is the standard normal bridge state. The Agent does not need to do anything to cross.
CLOSED – This is a state where the Agent needs to do some special Action to open it, but cannot pass through at the moment.
LOCKED – A locked bridge is beyond the ability of an NPC to open on his own, but may become open at some later time.



- All bridges that pass through entities like these can only connect through ONE entity
- Origins of bridges that connect through entites must be less than 150 units
- Breakable Brushes become "Open" when they are destroyed
- Func Walls will become "Open" when they are used (use again to lock)
- Func NPC Walls are the same as Func Walls except that they only stop NPC movement
- For double doors, it's best to have points on either side of each door, not down the middle two.



The Open, Closed, Locked states are used by all bridges. If a door is Closed, the agent will pause in front of it for a moment, play his "use" animation for that door / entity, and walk through when the bridge becomes open again.

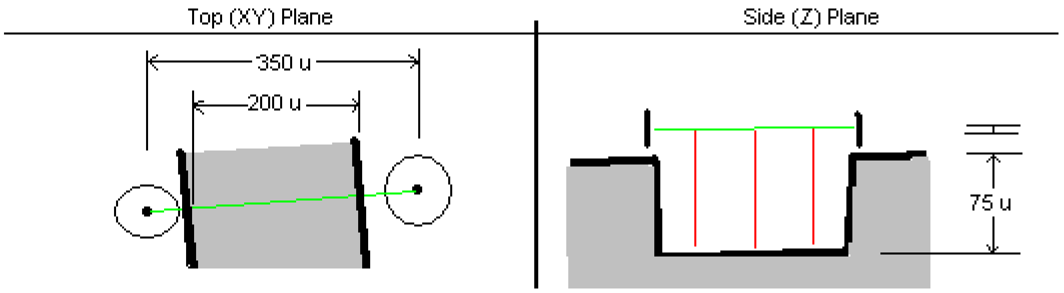
Entities that "Start Open" will be closed momentarily during the Nav grid connection process so that bridges can detect their existence, and will open up 50 milliseconds later.

Special Nav Point Connections

There are a number of cases that fall outside the normal connection rules and yet we still want agents to traverse these bridges.

Gap Jumping
In order to leap across a hole (a drop of more than 75 units with sharp walls on either side):

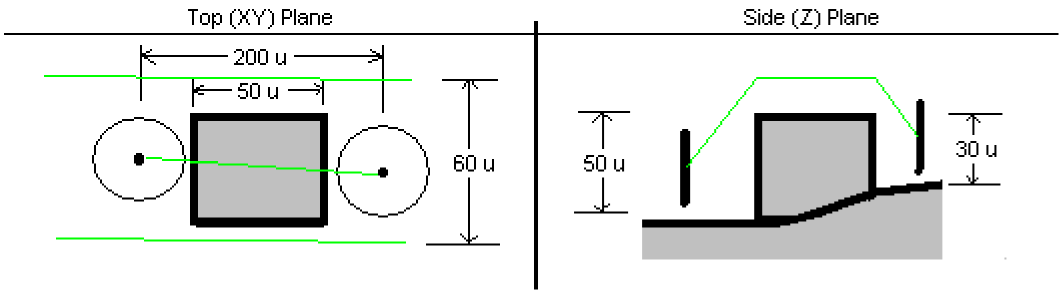
- Points must be less than 350 units apart
- Edges of hole must be less than 200 units apart along line of connection between Navs
- Hole must be detectable by one of 4 drops along connection line (red lines are drops)
- Points on either side must be no more than 5 units apart in height (Z Plane)



In most cases, just creating points on either side of a gap that is jumpable by the player should fall within these rules.

Ledge Vaulting
Vaulting over a ledge is perhaps one of the least likely situations to occur naturally without designer intention because the rules are very specific:

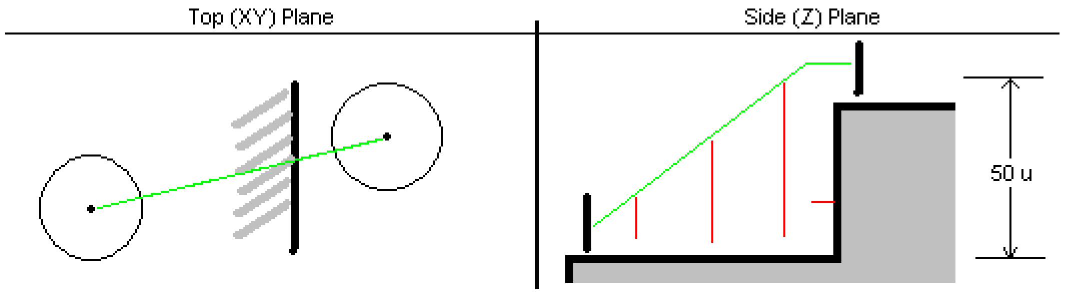
- The points cannot be more than 200 units apart
- The ledge must not be wider in the XY plane than 50 units
- The ledge cannot be higher on the highest side than 50 units
- The ledge cannot be lower than 30 units on the shortest side
- A 60 unit wide bounding box must be able to pass over the ledge



These rules may sound very specific, but I think if you just make a thing that is less than 50 units square and put points on either side, you should be fine.

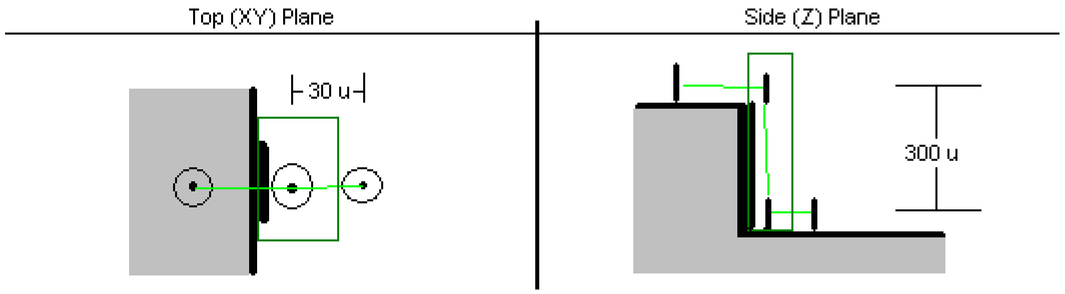
Drop Falling
Jumping off a ledge and landing below is an example of a one way dynamic connection.

- Points need to be more than 5 units apart in the Z axis
- There must be hole of more than 50 units somewhere along the connection (or it will be stairs)
- A trace from the lower of the two points must hit a vertical wall near the higher point



Ladder Climbing
Creating a ladder bridge is as simple as putting one at the bottom of the ladder and one at the top. If the ladder is more than 300 units tall, you will need one in the middle too.

- Points that connect to the top and bottom of the ladder from the rest of the grid need to be within 30 units
- Points inside the ladder need to be close enough to a solid brush (presumably the ladder itself) to find the wall (100 unit radius)
- The brush should go all the way to the edge of the ground at the top of the ladder so the guy does not end up in open air for any moment



Rappelling
1) Wall Bounce
The wall bounce works very similar to a ladder in that the point needs to hit a wall within 100 units. Wall bounce points will move themselves exactly 25 units away from the wall to match the animation.

2) Straight Slide Down
The difference is that straight slide points do not hit a wall and are free floating.

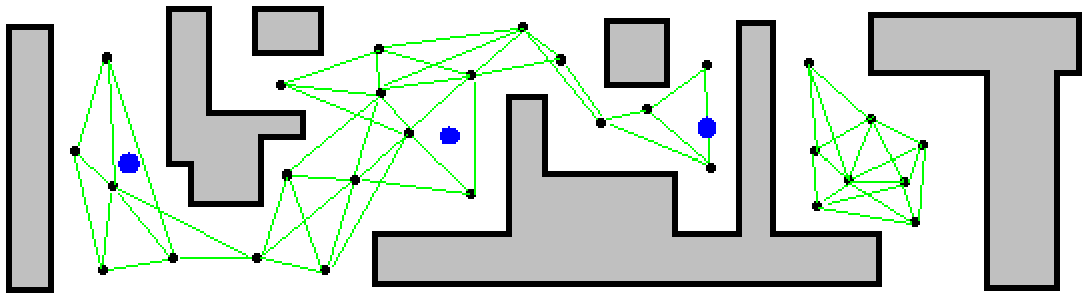
- The only rule specific to Rappelling is that the points must be almost exactly on the same XY coordinate (directly above or below each other) in order to connect.

Collections Of Nav Points

It is often important for characters to know more global geometric information about the world around them than a simple piecewise linear path. For these cases, there are two fundamental groupings of navigation points, Regions and Areas.

Regions

A region is a collection of points in which it is possible to reach any point from any other. In the diagram below, there are two regions. Path finding will check to make sure that the origin and destination points that were given to it are from within the same region before even attempting to find a path.

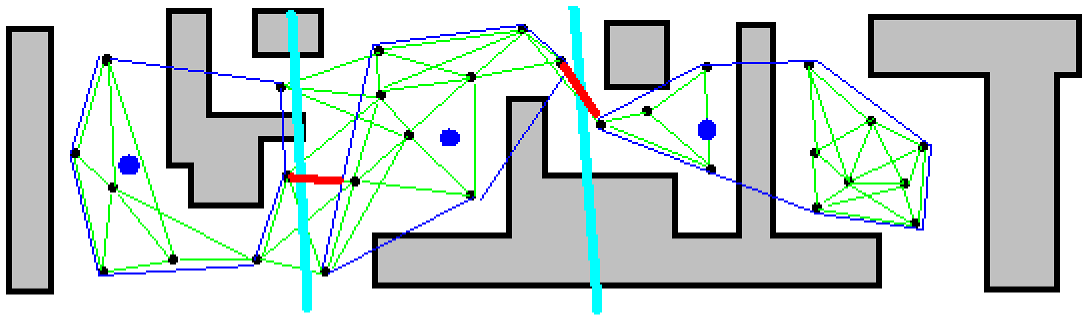


Regions can also be helpful during development because they are designed to uncover breaks in the grid. While some breaks are intentional or unavoidable, it's usually recommended to have as much of the map as connected as possible to allow characters to follow and not get stuck. The console will report any regions with less than 10 points in them during the level load time for reference purposes.

Areas

Where regions give you global connectivity information, Areas give you global geometric information.

Areas begin with a point that is created in very much the same manner as navigation points. Area points will show up in the game as very tall blue lines, and you can seen them on the diagram below as big blue dots. Then all the navigation points on the map are sectioned off and stored in their nearest area point. You can imagine this process as drawing a line that is exactly half way between neighboring areas and collecting all the points on each respective side (shown below with very light blue fat lines). This process is entirely 2D, so that areas will reach down through floors.



Another interesting point with this process is that it does not care at all about connectivity information. In the diagram above, you can see that the far right area actually includes points on two different sides of a wall and does not care in the least that they cannot be reached from either side.

Area Perimeters
Once all the points have been collected under their respective areas, we employ a standard computational geometry algorithm called "Graham's Scan" to find what we call the area's *Perimeter*. In computational geometry it's called the convex hull of a set of points.

A convex hull gives you the most extreme edges that still contain all the points and guarantees convexity. That way it's very easy to test any given world coordinates relative association to the area (in or out).

In the areas diagram, you can see perimeters as thin dark blue lines. In the game they will show up as fat white lines.

The details of how Graham's Scan functions and the properties of a convex hull object are discussions for another document, but if you are interested…

http://www.sit.wisc.edu/~cdreed/CHULL/CHULL.html
This is an animated Java applet of the exact same code we use to produce our Convex Hull.

http://riot.ieor.berkeley.edu/riot/Applications/ConvexHull/CHDetails.html
This site does a nice job of explaining the principals of GS.

Area Bridges
Finally, a simple search of all the points on all the perimeters will record the shortest navigation bridge between any two areas. Knowing this information serves as a guide to global movement patterns and can provide a heuristic to potential ambush locations, pinch points, and other various tactical knowledge.

In the areas diagram, you can see the Area Bridges as fat red lines. In the game they are also fat red lines.

Appendix

<u>Nav Point System Related Console Commands</u>

MOVEMENT

- gotonp <#>:	Will teleport player to the given point's position
- gotonpname:	Will teleport player to given point's position

AREAS

- makeap:	Creates a new area at player's position
- nameap:	Names the closest area point

CREATING POINTS

- makenp:	Creates a new nav point at player's position
- namenp:	Names the player's closest nav point
- labelnp:	Labels the ground cover type for the player's nearest nav point
- deletenp:	Removes the nav point and reconnects entire grid
- movenp:	Takes player's nearest nav point and repositions it at player's position

CONNECTING & SAVING POINTS

- connectnp:	Causes a full scale clean of the entire grid and reconnects everything
- savenp:	Saves the nav points to a file

- playernp:	Resets the player's nearest nav point

- g_NavPointDebug:
    1 - normal view (views all neighbors of players assumed nearest NP)
    2 - extended view (all neighbors and all neighbors neighbors of nearest NP)
    3 - normal view of NON-ASSUMED NP, forces NP
    4 - extended view of non assumed NP, forces NP