Polygon Soup for the Programmer’s Soul: 3D Pathfinding

Patrick Smith

1. Introduction

One of the fundamental goals of an AI system is to avoid making the unit appear "dumb." At the root of this challenge lies one of the hardest problems to overcome efficiently and believably: pathfinding. Today, 3D graphics and sound technologies begin to reach a level of realism that is easily destroyed by seeing an AI unit walk face-first into a solid wall, slide along the perimeter and ultimately get stuck on an errant polygon. Traditional technologies that worked well for games a few years ago fall apart when faced with the complexity of today’s 3D environments.

This paper addresses the pitfalls of attempting to pathfind the arbitrary world we call "polygon soup." It covers automatic data generation and compression, a run-time distributed algorithm, organic post-process modifiers, and solutions for tricky situations such as doors, ladders, and elevators.

2. Where to Get Good Data

There are numerous good algorithms for determining a path given a connectivity graph. For simple 2D games the connectivity graph was a side affect of the tiling system used to create the map. For 3D games, especially those constructed from arbitrary geometry, no such simple solution exists. So the problem exists, where do we get good data?

Imagine importing the outline of a stained glass window into your favorite paint program. The image consists of large, irregularly shaped regions of pure white space. Your task: color each of these regions to complete the stained glass window. Now, which tool is best for the job, the pencil (which would require you to color each and every pixel independently) or the paint bucket (which can recursively flood a whole region with color)? Should be an easy decision, right? So why is it that many level designers are forced to manually populate their levels with numerous pathfind nodes, sectors, grids, zones, planes, subdivisions, portals or whatever whoseewhatsit is popular that day? Now, visualize an automated system that could discover all possible traversable locations in your most complicated environment with a single click of the mouse. Can you picture it? Great! Let’s build it.

To solve this Herculean task we’ll borrow a simple idea from our paint program metaphor, namely, the recursive flood fill. The only tools we’ll need are a good collision detection system and the knowledge of a single point in the polygon soup where a unit can stand. The algorithm works as follows:

Use the collision detection system to determine if the unit can exist at the start point. If the area is valid, add that point to a list. This will be our task list. Now, loop over all the entries in the task list (which is currently one). For each entry in the list, simulate the unit taking a step in each of the four cardinal directions (north, east, south, and west). If one of these points passes the collision check, add it to the task list. Retain the connection information between these two points (point A can walk to point B).

There, you're done, once this algorithm finishes you'll have a complete connectivity graph of the current game level. Of course there are always details…

- **Detail #1: Of shapes and areas.**
  When doing the collision checks we aren't really dealing with points, but volumes. This is because we are attempting to simulate an object, and objects have volume. We need to decide what volume yields the best results. Different collision systems use different representations for an object's collision volume; some use cylinders, some spheres, while others use boxes. In order to cover space continuously, without leaving gaps between areas, we'll use boxes.

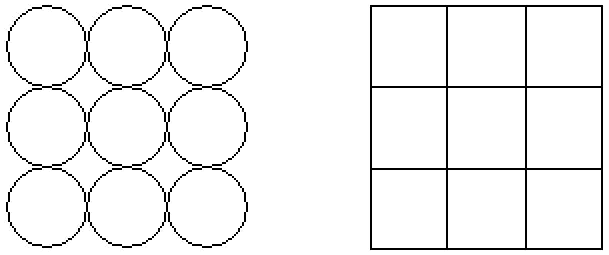
  

  Figure 1: Circular shapes form gaps when arranged adjacently, while boxes cover space continuously.

  At this point you should begin to see the familiar form the data is taking, a grid! However, this isn't the simple 2D grid you may be imagining. If you ignore the elevation of each box, the grid is uniform, however because we simulate walking from one grid cell to the next, it can take us up or down slopes, across bridges and over *and* under overpasses. This means that for any given coordinate on the "plane" there can exist more then one grid cell (consider elevation). A good example of this is a spiral staircase. The algorithm will fill up and around each turn of the staircase as if it were on a flat plane, however if you stood at the top and cast a ray down the course of the staircase you would encounter a grid cell at every turn of the stairwell.

- **Detail #2: Size does matter.**
  It is very important that the cell size is appropriate for the object pathfinding the level. Too large and the object will not be able to path through doors that are not orthogonal, too small and you'll find your computer running out of RAM before it finishes flood filling. As a suggestion, make an axis-aligned bounding box from the object's collision volume. This box should completely contain the object regardless of its orientation. Now, divide the X and Y dimensions by 2, thus giving you a "tall, skinny" box.

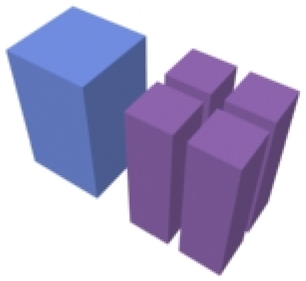


Figure 2: A good sized cell is generally a quarter of the object's maximized collision box.

The reason for this derives from the fact that the grid is axis-aligned (non-rotated). If a door or other passageway is rotated, and during game play the object fits through the door because it rotates as well, then a non-rotated pathfind cell will collide with the door.

- **Detail #3: It's not what you know, but who you know.**
  At a minimum, each cell of the grid only needs to know which neighbors are traversable. This can be accomplished via a simple data structure that has four pointers, one to each of its neighbors. If a pointer is NULL, that direction cannot be traversed.

  ```
  class CPathCell
  {
        .
        .
        .
        CPathCell * NorthCell;
        CPathCell * SouthCell;

        CPathCell * EastCell;
        CPathCell * WestCell;
  };
  ```

  This simple data structure completes a connectivity graph – we now have enough information to pathfind between any two cells in the level.

- **Detail #4: No bonus for extra work.**
  Be careful to avoid reprocessing cells that you already know are traversable. For example, we start with cell A and successfully simulate stepping to cell B. When it comes time to process cell B, we already know cell A has been processed so it should not be added back into the list. However, we still need to simulate stepping from cell B to cell A – just because we can walk from A to B does not guarantee we can walk from B to A (i.e. stepping from A to B takes you off a small cliff).

- **Detail #4: Isn't this a lot of memory?**
  Simply put, yes it is. Therefore it is extremely important to keep your data structures as tight as possible. For instance we've seen upwards of 10 million cells generated for a large level.

Keep in mind that this is a preprocess step, we're simply generating the data. Before we can use this data it must be compressed into a format that is feasible for run-time application.

3. Doors, Elevators, and Ladders – Oh my!

It is extremely satisfying to give an AI the command, "Goto (x,y,z)", and watch as the unit performs a complex series of steps. For example, given a single destination, an AI could walk to the front door of a building, open the door, walk to the elevator, wait for the elevator to arrive, take the elevator to the roof, exit the elevator, walk to the ladder at the base of the tower, and climb to the top of the tower.

Even more satisfying is to have this information about the "teleport" mechanisms automatically detected during the data generation step. To arrive at this goal, we'll need to know the location of each door, elevator, ladder, or other teleport mechanism, and it's enter and exit zones. The algorithm now works as follows:

During flood fill, grab an unprocessed cell from the task list. Perform the simulation as before, if the cell passes, check to see if it is completely contained within the enter zone of one of the teleport mechanisms. If this cell is indeed inside an enter zone, create a new cell in the center of the exit zone and perform a collision check to ensure it is safe to stand there. If the new cell passes this test, add it back into the task list – this will ensure the simulation continues on the other side of the mechanism.

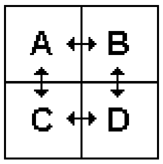
Avoid the temptation to store the connection between the "enter" cell and the "exit" cell. This extra data will unnecessarily bloat the size of each one of the cells in your 10 million cell flood fill. The connection information can easily be determined after the flood fill is complete. This issue is addressed later when discussing data compression.

4. What to do if you can't devote 500MB of RAM to Pathfind

Now that we have a nice simple algorithm for generating the connectivity graph of a level, it is time to start inspecting the data. Let's say each cell requires approximately 50 bytes of data, and we've generated somewhere on the order of 10 million cells, that means we have 500MB of pathfind data. When problem solving, the more data the better, but half a gigabyte is just plain ridiculous! It's time to compress the data down to something more manageable.

Let's examine a simple scenario. If cell A connects to cell B, and cell B connects to cell A, then do we really need two cells? No we do not; we can combine the two cells into a *sector* and throw away the original cells. We define a sector to be a convex polyhedron of freely traversable space. Given our particular dataset and purpose, a box is the most effective convex polyhedron.

To expand on this compression scheme, consider the following diagram.



Cell A connects to B and C, cell C connects to A and D, cell D connects to C and B, cell B connects to D and A. Here, we can combine all four cells into a single sector and discard the original cells. To generalize, we can combine any rectangular set of contiguous, connected cells into a single sector. Ultimately, we'll want to compress all the cells into sectors. The sector then becomes the basic data type used during runtime to solve a path.

Once a sector is generated, it is necessary to retain the connection information its composite cells originally contained. To accomplish this, retain the edge cells and discard the interior cells of the sector. These edge cells share connection information with cells in other sectors, in effect, linking sectors together.

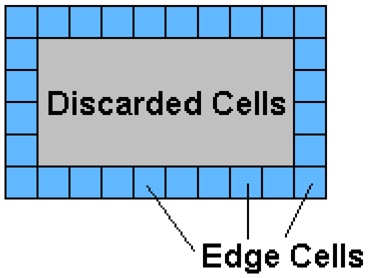


Figure 3: To retain a sector's connection data, keep the edge cells.

After all the sectors have been built, you can then combine a sector's edge cells that share a common destination sector into a *portal*. We define a portal to be a connection between sectors; i.e. to get from sector A to sector B walk from your current position in sector A through the AB portal. The portal's physical shape is the bounding volume of its composite edge cells.

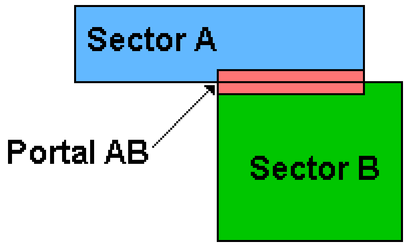


Figure 4: Portals are connections between two sectors.

Portals can be one-way or two-way. If the edge cells in sector A are connected to the edge cells in sector B, and the same cells in sector B are connected to the sector A cells, then the portal is two-way. Otherwise the portal is one-way.

Once the portal is created, it is added to the list *portal-list* of the sector that contained the edge cells. If this is a two-way portal, it is added to the portal-list of both sectors.

5. Doors, Elevators, and Ladders – Again!

Earlier we discussed how to flood fill across teleport mechanisms such as doors, elevators and ladders. Now that we've converted all our cells into sectors and portals, it's time to incorporate information about these mechanisms so that our AI can use them.

To build a portal for a given mechanism, first intersect its entrance and exit zones with the existing pathfind sectors. If the intersection test yields at least one sector for the entrance and exit zones then we can create a portal between the two. The entrance portal's bounding box is the volume created by intersecting the entrance zone with the pathfind sector. The exit portal's bounding box should follow the same formula.

Store any specific information about the mechanism the AI will need to know in order to operate it in the portal. For example, if a door is locked, save the type of key that's needed to open the lock. When the path solver is attempting to resolve the path, it can ignore the portal if the AI is not carrying the required key.

6. Anatomy of a Sector

Once a sector is generated it contains two important pieces of information, a bounding box and a list of portals. The bounding box is used at run-time to determine where, in the pathfind world, an AI currently is. The portal-list is the connection information that the path solver uses to calculate how an object gets from its current position to its destination position.

```
class CPathfindSector
{
      .
      .
      .
      CAABox                        BoundingBox;
      CVector<CPathfindPortal *>    PortalList;
};
```

As you can see, the sector is really just a spatial portal linkage; most of the interesting data is contained in the portal.

7. Anatomy of a Portal

The portal contains as much data as is necessary to get the AI from one sector to the next. At a minimum this data is a reference to the destination sector (possibly two sectors for a two-way portal), and its bounding box. A portal may also contain information about a teleport mechanism or an action to perform.

```
class CPathfindPortal
{
      .
      .
      .
      CPathfindSector *       DestSector1;
      CPathfindSector *       DestSector2;
      int                       MechanismID;
      MECHANISM_TYPE            MechanismType;
      CPathfindPortal *         MechanismExitPortal;
      ACTION_TYPE               ActionType;
};
```

As you can see all the connection information is stored in the portal and is therefore the "key" to solving a path.

8. Solving the Path

Bob, the digital super spy, needs to avert international tragedy at the UN building. Unfortunately, he's currently washing dirty socks in his basement. To make matters worse, everywhere he looks he sees an ocean of polygons swimming before his eyes. How can he possibly hope to reach his goal before time runs out?

Luckily for Bob, we can use his (X,Y,Z) location to lookup the pathfind sector he's washing socks in. This sector contains a list of portals he can use to get to other sectors, which contain their own portal lists, which can be used to get to yet more sectors, and so on and so forth.

At this point any number of best-path algorithms can be applied to get Bob to the UN building. We found a modified A-star algorithm solves the problem nicely. A simple implementation would use the accumulated distance between portals as the traversal cost, and the straight-line distance to the goal as the heuristic cost.

Now pretend that Bob and twenty of his closest buddies need to get to the UN building at the same time. Twenty different requests for complex paths can well exceed the amount of CPU your game devotes to AI each frame. Luckily for us, it is not very noticeable if we distribute the processing of these paths over a few frames. For example, if Bob sits and thinks for a half second before getting up and walking out the door, is anyone likely to notice? On the flip side, a half a second (500 milliseconds, ~1 billion clock cycles) is more then sufficient for a pathfinding system to solve a few simultaneous paths.

To distribute path solves over multiple frames you could use an algorithm like the one outlined in the following pseudo-code.

```
while (Time_Has_Not_Elapsed () && CurrentPathSolver != NULL)
{
      if (CurrentPathSolver->Process (Time_Remaining_This_Frame))
      {
            RELEASE (CurrentPathSolver);
            CurrentPathSolver = Get_Next_Path_Solver ();
      }
}
```

This is a simple time slice manager which allows the current path to process a little bit each frame. When the path is solved, a new path is processed until either it finishes or time has elapsed.

To distribute the path solve itself over multiple frames, consider the following pseudo-code.

```
CPathSolver::Process (float time)
{
      while (Time_Has_Not_Elapsed)
      {
            CPathfindSector * sector = Get_Least_Cost_Sector ();

            if (sector == NULL)
            {
                  Handle_No_Path ();
            }
            else if (sector == DestinationSector)
            {
                  Hande_Path_Found ();
            }
            else
            {
                  CPathfindPortal *portal = NULL;
                  while (portal = sector->Get_Next_Portal (portal))
                  {
                        Process_Portal (portal);
                  }
            }
      }
}
```

This algorithm will process as many sectors as necessary until its time-slice has elapsed.

9. Loosen Up Dude! Avoiding Robotic Paths

Using the sector/portal system, a solved path is represented by an ordered series of portals. This path will follow the form: walk to portal AB, turn to face portal BT, walk to portal BT, turn to face portal TU, walk to portal TU, and so on. Unless your AI is a robot, it is desirable for the unit to behave as organically as possible; bee-lining to each portal does not make for a very organic path.

To make our path appear a little more natural we'd like to use splines. In layman's terms, splines are basically curved lines that follow a set of points. Different types of splines follow their points in different ways. Some splines pass through their control points, whereas other splines act like magnets and are either attracted or repulsed by their control points. To see the advantage of splining the path, consider the following diagram.

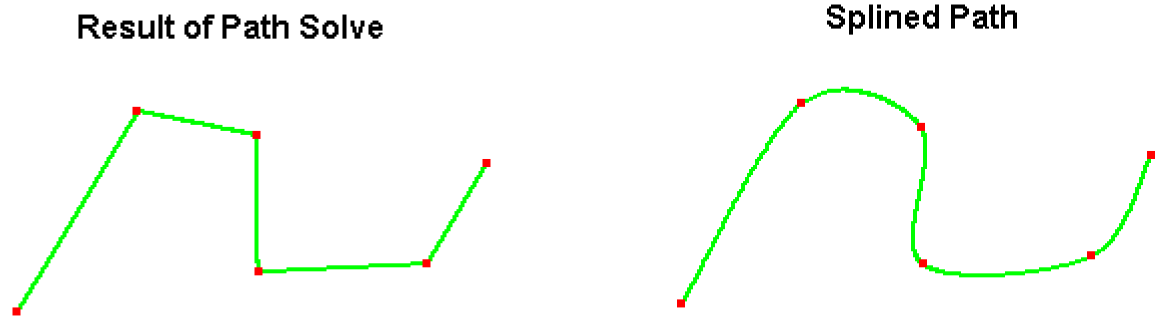


Figure 5: Splining the final path yields a much more organic curve.

Looks great, right? Unfortunately, there is a problem. Since the spline doesn't know anything about the sectors and portals that define the "safe" area for a unit to walk, it is likely that a curve on the spline will leave these volumes causing the unit to walk into desk corners, doorways, trash cans, or even meander off the edge of a cliff!

Luckily we can take advantage of a mathematical property of Bezier curves (a type of spline): if the control points of the Bezier curve lie inside a volume, then the curve itself will lie inside the volume. A Bezier curve is a cubic spline defined by a start point, an end point and a series of control points. The control points act like magnets to "pull" the curve away from the line segment defined by the start and end points.

Given this, our organic post process algorithm would function as follows:

Build a Bezier curve from the points on the final path. Build a list of the sectors and portals that the AI will pass through on its path. For each control point on the Bezier curve, clip it to the sector volumes. To clip a control point, form a line from the control point to the corresponding point on the *un-splined* path. Test this line segment to see if it passes through the side of any of the sector boxes in the list. If it does, check to see if the area of intersection is a portal. If the point intersects a sector wall, and it does not pass through a portal, then clip the control point to the sector wall; otherwise leave the point alone – it is valid.

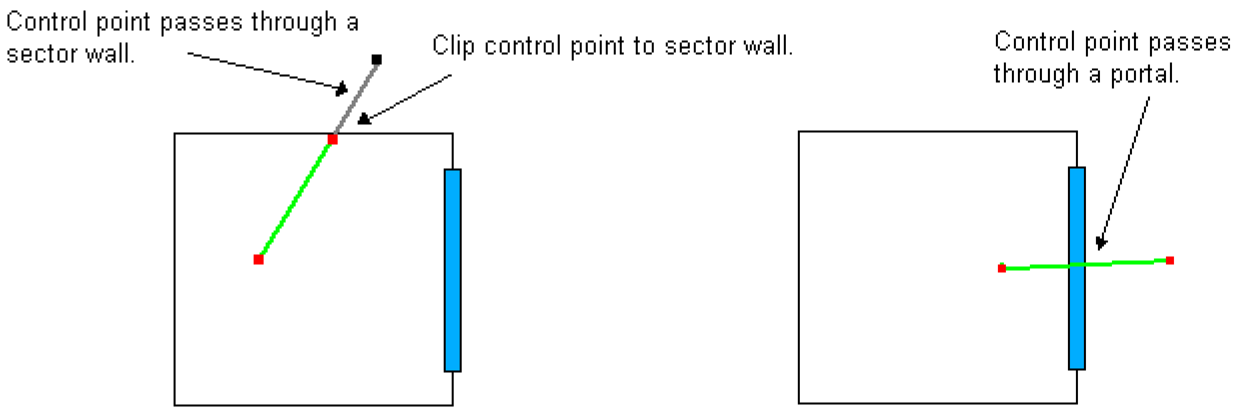

Control point passes through a sector wall.
Clip control point to sector wall.
Control point passes through a portal.

Figure 6: Control points that pass through a sector wall need to be clipped. Control points that pass through a portal are valid.

This algorithm will allow the AI to follow a nicely curved path without ever leaving the safe pathfind sectors.

10. Using Innate Waypaths

In a busy city, it would be odd to see cars driving down the sidewalk and pedestrians meandering through the streets. However, using our current pathfinding system this is exactly what you'd see. The flood fill algorithm does not encode (nor is it aware of) the type of ground covered, be it sidewalk, highway, or burning hot lava. A possible solution to this problem is to store surface information in each cell of the flood fill and have your compression algorithm only generate sectors containing cells of the same type. This would yield sidewalk sectors, highway sectors and burning hot lava sectors. The run-time path solver could then utilize this information when determining what sectors are valid for a given object (i.e. cars only pathfind through street sectors).

Innate waypaths are another powerful solution that integrates well with our pathfind system. A waypath is a manually created series of waypoints which form a path. This waypath can be rigid or splined. Each waypoint on the waypath can encode specific information, such as crouch here, speed up, slow down, or even jump to the next waypoint. To integrate this information into our path solver, consider the following algorithm.

Run a normal pathfind flood fill to generate sectors and portals. For each innate waypath, create a “dummy” sector (a sector without size or position), and add it to the system. For every waypoint in the waypath, find which pathfind sector the waypoint intersects. Create a two-way portal from the pathfind sector to the dummy waypath sector at the location of the waypoint. Add this new portal to both the pathfind sector and the dummy waypath sector.

During the run-time path solve, the algorithm’s heuristic should bias toward these waypath sectors (i.e. multiply the cost of using a waypath sector by 0.75). This will cause AIs to “tend” to follow innate waypaths.

Given this system, to enable vehicle AIs to drive along the right side of the street, a level designer would simply draw a one-way, vehicle only, innate waypath along each side of the road.

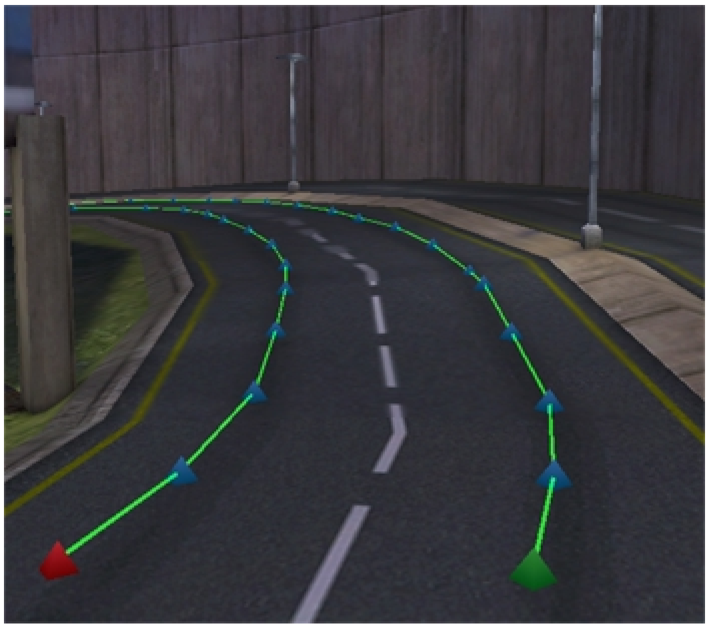


Figure 7: Innate waypaths can be used to keep vehicles driving along the right side of the street.

11. Follow the Leader

Currently, players are smarter then AIs: humans have the ability to combine abstract concepts and make leaps of logic. This reasoning ability allows players to perform actions that AIs cannot originate, but if we're lucky perhaps they can replicate. For instance, during a firefight on the second floor of a building, the player reasons he cannot win. In desperation he shoots out the window and, with a mighty leap, escapes onto the street. What does the AI do? Unless we give it an option, the AI will pathfind down the stairs, open the front door and look dumbly down the street, for the player is long gone. Wouldn't it be much more fun if the AI simply jumped out the window after the player?

There are many different ways to approach this problem, so let's choose the simplest. We know where the player jumped from and we know where the player landed. If both points are inside the pathfind data, why not add a temporary one-way "jump" portal between these two sectors? All we need to do is encode a little information about the jump such as orientation, velocity, and time. Depending on the physics system, this may be enough for the AI to replicate the player's mighty leap. We can even keep a FIFO "bucket" of these temporary portals so other AIs can follow.

12. Vehicles

Arbitrary vehicle pathfinding is an order of magnitude more difficult then traditional bipedal pathfinding. Humans can turn on a dime, whereas vehicles have a turn radius. Humans follow the same rules regardless of orientation, whereas vehicles act differently in drive and reverse. Humans can stop on a dime, whereas vehicles skid. Humans do not tip over when running too fast, whereas vehicles can roll.

A complete vehicle pathfinding solution is outside the scope of this paper, however we will present a brief overview that may work for some games.

Firstly, modify the flood fill algorithm to use the bounding volume of the largest vehicle regardless of its orientation. During the run-time path solve, take the turn radius of the vehicle into consideration when evaluating portals. In other words, we know which portal the vehicle is coming from, so discard any destination portals that would cause the vehicle to turn sharper then it is able.

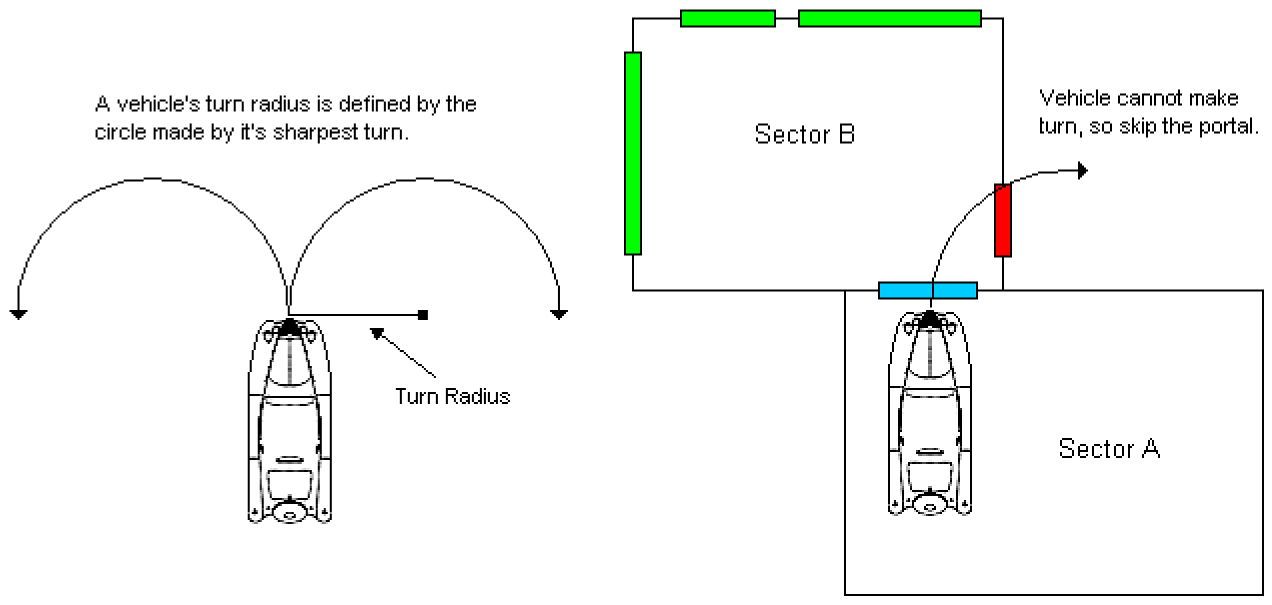

A vehicle's turn radius is defined by the circle made by it's sharpest turn.
Turn Radius
Sector B
Vehicle cannot make turn, so skip the portal.
Sector A

Figure 8: Take turn radius into consideration when evaluating portals.

Once the path is generated, it may be problematic for a vehicle to follow. This is because orientation is important when dealing with vehicles; however our sector/portal system doesn't implicitly handle this very well. Consider the following diagram.

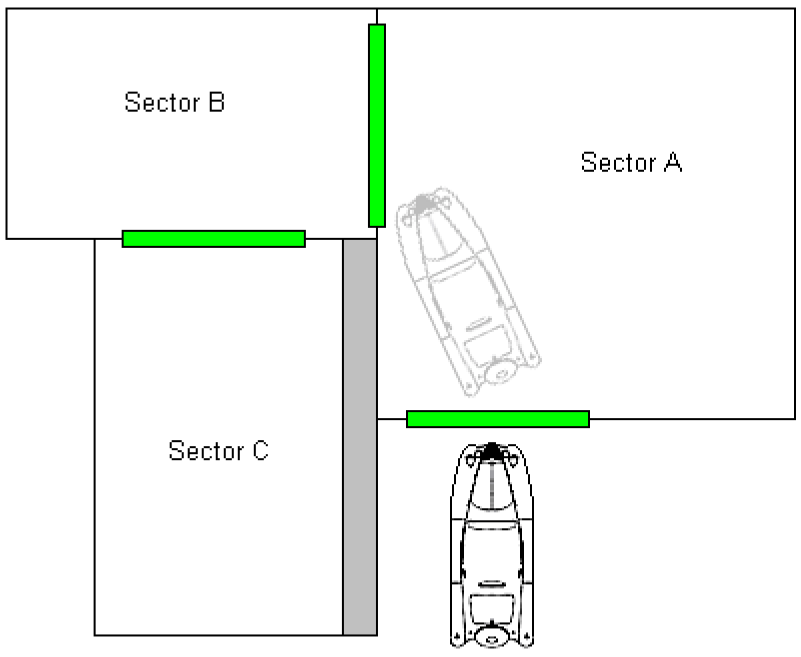


Figure 9: Orientation is important when traversing the path. Once in Sector B, the vehicle will never make the turn to Sector C.

In the sector/portal system, the vehicle will beeline from the Sector A's entrance to Sector B. This orients the vehicle in such a way that it will be impossible for the vehicle to make the following turn into the portal for Sector C. However, if the vehicle would "arc" out into Sector A, it would be possible to make both Sector B and Sector C. Unfortunately there is no way of knowing this is required unless we search ahead on the path.

A simple solution to this problem, which works well with the system we've described so far, is to spline the path. However, none of the classical "true" splines will work for this situation; which is fine -- we'll create our own custom curve.

Our goal: create a continuous curved line that will cause the vehicle to arc around corners while still obeying the vehicle's turn radius restrictions. Such a path can be created using only a straight line and the vehicle's turning circle. Unlike "true" splines, this path is not continuous in the mathematical sense, but composed of three distinct continuous parts, which, when placed end-to-end, form a continuous path. The three parts are: the exit curve from the previous point, a straight line from the exit curve to the enter curve of the next point, and the enter curve of the next point. Note: The starting and ending points only contain two parts (there is no previous part for the starting point, nor is there a next point for the ending point). Consider the following diagram:

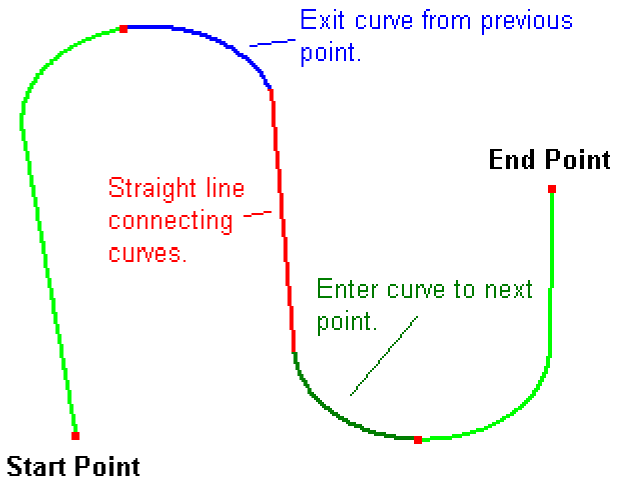

Exit curve from previous point.
End Point
Straight line connecting curves.
Enter curve to next point.
Start Point

Figure 10: The vehicle curve is composed of three distinct parts: the exit curve, a straight line, and the enter curve.

To build this curve, overlay the vehicle's turning circle onto each node of the path. We will assume the optimal center of this turn arc will lie at the point halfway between the angles formed by the (prev_node – curr_node) and (next_node – curr_node) vectors. This causes the actual node point to lie on the perimeter of the turning arc.

For each turning circle on the path, find the "in" and "out" tangent points. The "in" tangent point is the closest point of tangency from the previous point to the turn arc. The "out" tangent point is the closet point of tangency from the next point to the turn arc. Now, simply connect the dots. The path is as follows: straight-line from starting point to the “in” tangent point on the turn arc of the next path node; follow the turn arc to “out” tangent point; straight line from this point to the “in” tangent point of the turn arc of the next path node, etc, etc.

Once finished, this algorithm yields a continuous curve that follows the turning restrictions of the vehicle following the path.

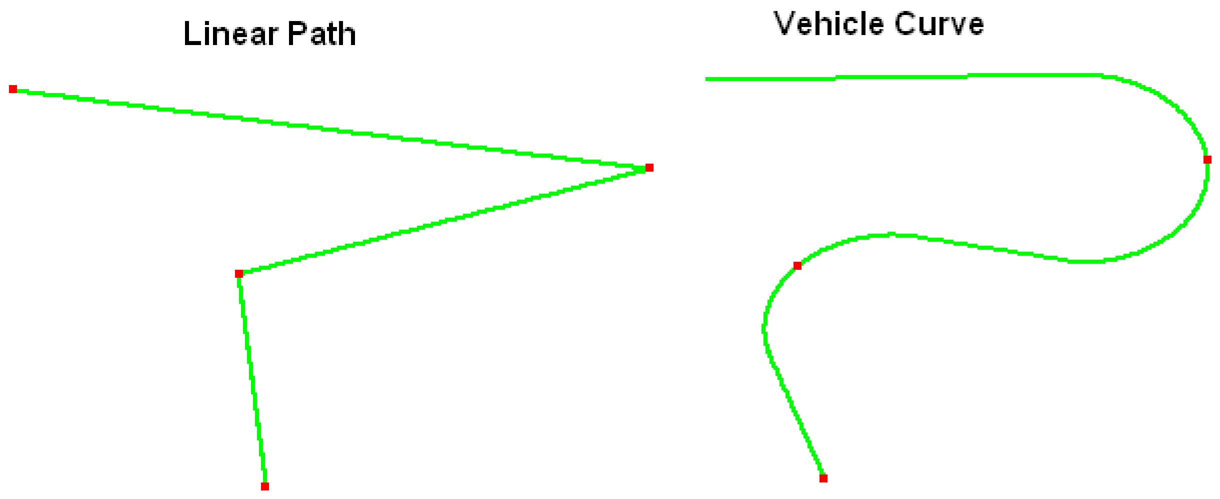


Figure 11: The vehicle curve ensures the vehicle will not attempt any impossible turns.

Keep in mind that this curve may cause the vehicle to drive outside the “safe” areas of the path, thus potentially colliding with objects along the way.

13. Conclusion

By taking advantage of a simple flood fill algorithm, we can overcome the Herculean task of automatically generating connectivity data through complex polygon soup. With a few tricks and extensions we can easily incorporate doors, ladders, and elevators; spline the result of the path solve to yield a more organic looking path; dynamically alter the pathfind data to allow the AI to replicate a player’s actions; incorporate innate waypaths to force a specific behavior from the AI; and distribute the run-time path solve over multiple frames to balance the CPU load.

Special thanks to Eric Cosky for the original concept of the flood fill algorithm, a brilliant idea that proves the worth of brainstorming with a friend before jumping into a complex problem. Also, thanks to Colin Mclaughlan for suggesting the concept of dynamic temporary “jump” portals.