3.3

The Basics of A* for Path Planning

Bryan Stout

The Problem

This article examines the basic solution to the problem of planning a path for an autonomous agent to move from one location in a game world to another, a common situation in computer game AI. On the CD that accompanies this book is a copy of my PathDemo program, with which you can play to understand how A* (and other path-planning techniques) works.

The most common issue involved in path planning is the avoidance of obstacles, including cul-de-sacs to be ignored (or, sometimes, discovered and explored). The next most common issue is perhaps the awareness of different terrain and seeking out the most efficient path among a variety of choices: exploiting roads or clear terrain, avoiding swamps, and so on.

An Overview of the Solution

The A* (pronounced *A star*) algorithm is an old workhorse in the academic AI community, used since 1968 for solving different kinds of problems, of which the 15-puzzle is the favorite teaching example; fortunately, it is also very useful for the path-planning problem.

A* is an algorithm that searches in a *state space* for the least costly path from a start state to a goal state by examining the *neighboring* or *adjacent states* of particular states. In the 15-puzzle, a state consists of a configuration of the 15 tiles in the 4×4 array, and an adjacent state is reached by sliding one tile into the blank area. In the path-planning problem, a state consists of the agent occupying a particular location in the game world, and an adjacent state is reached by moving the agent to an adjacent location.

In essence, the A* algorithm repeatedly examines the most promising unexplored location it has seen. When a location is explored, the algorithm is finished if that location is the goal; otherwise, it makes note of all that location's neighbors for further exploration.

In more detail, A* keeps track of two lists of states, called **Open** and **Closed**, for unexamined and examined states, respectively. At the start, **Closed** is empty, and **Open** has only the starting state (the agent in its current position). In each iteration, the algorithm removes the most promising state from **Open** for examination. If the state is not a goal, the neighboring locations are sorted: If they're new, they're placed in **Open**; if they're already in **Open**, information about those locations is updated, if this is a cheaper path to them; if they're already in **Closed**, they are ignored, since they've already been examined. If the **Open** list becomes empty before the goal is found, it means there is no path to the goal from that start location.

The "most promising" state in **Open** is essentially the location with the lowest estimated path that would go through that location. Each state *X* includes information to determine this: the cost of the cheapest path that has led to this state from the start (which we'll call `CostFromStart(X)`); a heuristic estimate `CostToGoal(X)` of the cost of the remaining distance to the goal; and the total path estimate, defined as `CostFromStart(X) + CostToGoal(X)`. The total path estimate is the lowest `TotalCost(X)` value that it determines is the next state to examine. In addition, each state keeps a pointer to its "parent" state, the state that led to this one in the cheapest path to it; when a goal state is found, these links can be traced back to the start in order to construct the path from start to goal. Please note that in the literature, you'll find `CostFromStart(X)` called g(X), `CostToGoal(X)` referred to as h(X), and the total path estimate named f(X). We'll use our names for greater clarity in this article.

Listing 3.3.1: The A* Algorithm

In pseudocode form, here is the A* algorithm:

```
Open: priorityqueue of searchnode
Closed: list of searchnode

AStarSearch( location StartLoc, location GoalLoc,
agenttype Agent ) {
    clear Open and Closed

    // initialize a start node
    StartNode.Loc = StartLoc
    StartNode.CostFromStart = 0
    StartNode.CostToGoal = PathCostEstimate( StartLoc,
        GoalLoc, Agent )
    StartNode.Parent = null
    push StartNode on Open

    // process the list until success or failure
    while Open is not empty {
        pop Node from Open      // Node has lowest TotalCost

        // if at a goal, we're done
        if (Node is a goal node) {
            construct a path backward from Node to StartLoc
```

```
            return success
        } else {
            for each successor NewNode of Node {
                NewCost = Node.CostFromStart + TraverseCost( Node,
                NewNode, Agent )
                // ignore this node if exists and no improvement
                if (NewNode is in Open or Closed) and
                (NewNode.CostFromStart <= NewCost) {
                    continue
                } else {        // store the new or improved
                                information
                    NewNode.Parent = Node
                    NewNode.CostFromStart = NewCost
                    NewNode.CostToGoal = PathCostEstimate( NewNode.Loc,
                    GoalLoc, Agent )
                    NewNode.TotalCost = NewNode.CostFromStart +
                    NewNode.CostToGoal
                    if (NewNode is in Closed) {
                        remove NewNode from Closed
                    }
                    if (NewNode is in Open) {
                        adjust NewNode's location in Open
                    } else {
                        push NewNode onto Open
                    }
                }
            } // now done with Node
        }
        push Node onto Closed
    }
    return failure   // if no path found and Open is empty
}
```

Properties of A*

A* has several useful properties. (They are not proved here; readers who are interested in the proofs can look in the References.) First, A* finds a path from the start to the goal, if one exists. Second, it finds an optimal path, as long as the `CostToGoal`(X) estimate is *admissible*, which means `CostToGoal` is always an underestimate—that is, `CostToGoal`(X) is always less than or equal to the actual cheapest path cost from X to the goal. Third, A* makes the most efficient use of the heuristic: No search that uses the same heuristic function `CostToGoal`(X) to find optimal paths examines fewer states than A*, not counting tie breaking among states with equal cost.

Applying A* to Game-Path Planning

Let's now look in detail at how the aspects of A* can be applied to path planning in computer games. Much of this discussion depends on the nature of the game and its internal representation of the world; the following discussion is meant to suggest possibilities.

State

As stated above, the principle component of a *state* in the path search is *location*. However, it need not be the only component. An agent's orientation and/or its velocity can also be important. For example, vehicles can often go only straight ahead or turn slightly, and the amount of turn possible is reduced the faster a vehicle moves. Most vehicles can go backward only after coming to a stop. It is quite possible to plan a route based only on locations, but it could be desirable in some situations to plan based on velocity and orientation as well, to avoid planning a route through terrain or around obstacles that would be difficult to navigate.

Even considering location alone, the issue of which locations to consider is not trivial. In some games, the world is naturally tiled—real-time strategy games often have an underlying square grid, and many war games use a visible hex grid—but many games do not divide the space that way, especially games that use a 3D, first-person, or oblique view of the world. In such cases, it is important to choose a set of locations among which to search. Figure 3.3.1 shows a path-planning situation and several ways of partitioning the space.

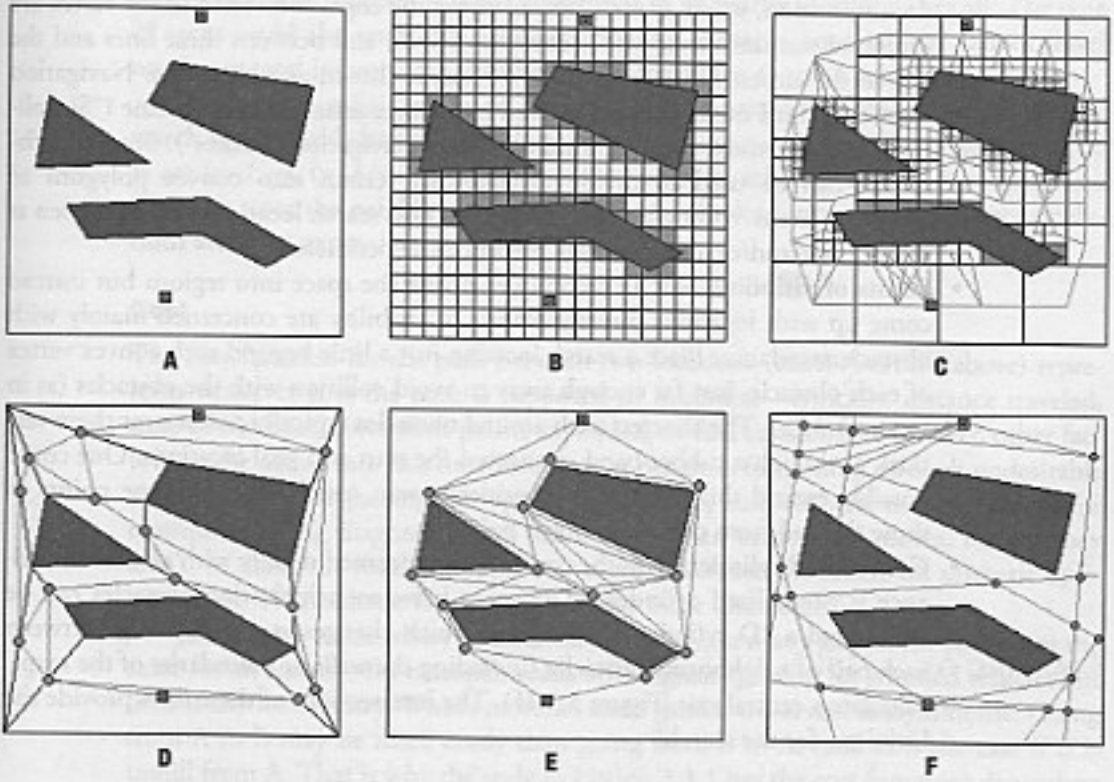

FIGURE 3.3.1. A variety of means of partitioning a continuous space.

The ways of partitioning the space are as follows:

- **Rectangular grid.** The simplest way is to partition into a regular grid of squares, as shown in Figure 3.3.1b. The locations can be either the center points or the corners of the squares; if appropriate for the game, the grid can be considered to consist of the terrain most common to the area it covers.
- **Quadtree.** Another way to partition the space is into squares of differing sizes. The quadtree recursively divides a square into four smaller squares, until each square has uniform (or at least mostly uniform) terrain, as shown in Figure 3.3.1c. Again, the locations for the search can be either the centers or the corners of the squares. This method has a couple of advantages: The larger (and fewer) squares allow for a faster search, and the representation is easy to store.
- **Convex polygons.** A more complex yet possibly more robust scheme is to break up the space into convex polygons made up of uniform terrain (Figure 3.3.1d). This scheme could already exist in the map's representation, so it can be used directly in the path search. There are several methods that can be used to partition a space into polygons if the existing mesh is useless or inefficient. C-cells are one way to partition the space; each vertex is connected to the nearest visible vertex, and the connecting lines partition the space. Another is maximum-area decomposition, where at each convex vertex the edges connected to the vertex are projected out until they hit an obstacle or wall, and between these lines and the line to the closest other vertex, the shortest is chosen as a boundary. Navigation meshes, a third method, are discussed in another article in this volume ("Simplified 3D Movement and Pathfinding Using Navigation Meshes"). Similar techniques can be used to divide variable-cost terrain into convex polygons of uniform terrain. After the polygons are laid out, search locations can be chosen at their center and/or along various parts of their perimeters.
- **Points of visibility.** Not all techniques divide the space into regions but instead come up with locations directly. Points of visibility are concerned mainly with obstacle avoidance: Place a search location just a little beyond each convex vertex of each obstacle, just far enough away to avoid collision with the obstacles (as in Figure 3.3.1e). The shortest path around obstacles typically passes near these vertices, as though a rubber band connected the start and goal locations. One could possibly extend this method to consider terrain cost by adding these points to those derived from convex uniform polygons.
- **Generalized cylinders.** Another technique concerned mainly with obstacle avoidance is generalized cylinders: The space between neighboring obstacles can be considered a 2D cylinder, the shape of which changes as it goes along. Between each pair of neighboring obstacles (including the walls or boundaries of the map), calculate a central axis (Figure 3.3.1f). The intersections of these lines provide the locations for the search.

For most of these schemes, when a search is done, the start and goal locations are usually not members of the search locations, so they need to be added to them for the duration of that search.

Whichever scheme is used for quantizing a continuous space, it probably must be experimented with and tweaked before it suits the game's demands optimally: There need to be enough locations so that no reasonable route is unconsidered, but not too many, or the search will take too long. Another issue is that most paths found from any quantization scheme seem jagged and a bit artificial, which means the route needs to be smoothed, either before it is assigned to the agent or in the means the agent uses to follow it.

Neighboring States

The neighbors of a state are determined by the map representation and the quantization scheme. Some schemes use only their adjacent locations as their neighbors—a square grid would consider each interior point as having eight neighbors, four if diagonals are excluded—whereas in other schemes, a location's neighbors are all other locations visible to it.

A location's neighbors are also determined by the terrain. Some terrain might be impassable, which means it is not a neighbor to its nearby locations after all. The type of agent could also enter this determination; for example, land vehicles cannot travel on the sea, and infantry can traverse terrain forbidden to some vehicles.

We need an efficient way to compute each location's neighbors, for the sake of search speed. Grids have a natural way of calculating neighbors: The neighbors of (x, y) are $(x+1, y)$, $(x+1, y+1)$, $(x, y+1)$, etc. Most other schemes require that some data structure store the neighbor information for fast lookup, since the neighbor calculations are often expensive.

Cost

The cost function for the path between two locations (`CostFromStart` above) represents whatever it is the path is supposed to minimize—typically, distance traveled, time of traversal, movement points expended, or fuel consumed. However, other factors can be added into this function, such as penalties for passing through undesirable areas, bonuses for passing through desirable areas, and aesthetic considerations (for example, making diagonal moves more costly than orthogonal moves, even if they aren't, to make the resultant path look more direct; see the article on aesthetic optimizations, "A* Aesthetic Optimizations," for more discussion.)

Just as with connectivity considered previously, in many games, the cost is not the same for all agents—for example, roads offer a great speedup for wheeled vehicles but little if any for infantry. What's more, in some games, travel cost is asymmetric: Going from A to B may be more costly than going from B to A, such as is the case if B is uphill from A. That is why the code in Listing 3.3.1 has the cost functions dependent on the agent traveling as well as the two endpoints of the travel.

Again, these terrain costs need to be quickly looked up during the search and in fact are probably best stored with the connectivity information. In that way, one lookup can determine whether two locations are neighbors, and if so, the cost for the given agent.

Estimate

The estimate of the path cost to the goal is the complement to the known distance from the start. If you want to guarantee that an optimal path is found, this distance should not be overestimated. A common way to do this is to multiply the actual map distance from the given location to the goal times the minimum terrain cost per unit distance. Since the route cannot be shorter than the most direct, "crow's flight" line, this figure will be an underestimate (unless your game has things like instant-transport locations). In many games, this minimum distance is the Euclidean measure between two points in 2D or 3D, but in games with strict square or hexagonal tiles, the shortest tile path is usually a little longer than the Euclidean distance between the tile's center points. Therefore, in a square grid, a tile (3, 5) away has a minimum distance of 2 + 3*sqrt(2), not the Euclidean sqrt(34). This actual shortest distance can then be multiplied by a typical terrain cost. This cost should include all the previously discussed factors concerning the cost between neighboring nodes.

However, guaranteeing an optimum path is not the only consideration; there is also the speed of the search, and the quality of the `CostToGoal` value has a tremendous impact on the search efficiency. Look at Figures 3.3.2 and 3.3.3. In Figure 3.3.2a, the `CostToGoal` has been set to zero, which, after all, *is* an underestimate, and we see that the search spreads in a circle until it hits the goal, because it has no heuristic information to guide it in the correct direction. In Figure 3.3.2b, we see that an accurate heuristic weight of 1 per square sends the search in a straight line to the goal. In Figure 3.3.3, the start and the goal are in costly terrain (8 per square). Since the estimate of 1 per square is a large underestimate, the search frontier is nearly as circular as the uninformed search in Figure 3.3.2a. In Figure 3.3.3b, we see that even an estimate of 5 per square focuses the search considerably. It is therefore important, perhaps crucial, that the estimate be fairly accurate. In fact, in some situations, one might want to *overestimate* the cost to the goal in order to get a fast search, at the risk of getting a suboptimal path (the article "A* Speed Optimizations" talks about this concept). So rather than using a *minimum* terrain cost per unit, we could use a *typical* cost, which can either be fixed or dynamically determined by sampling the terrain between the stated start and goal.

Goal

The goal is typically a single location, but it does not have to be. For example, if a vehicle low on fuel is trying to plan a route to the closest refueling station, with each new node *N* in the search an estimate is made of the remaining distance to each station, and the minimum of them is used as the `CostToGoal`(N) value. This method

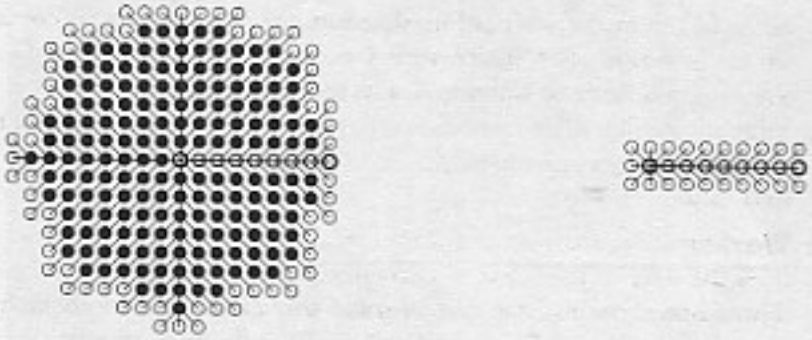

FIGURE 3.3.2. A* search in clear terrain. *a:* With a heuristic weight of 0. *b:* With a heuristic weight of 1.

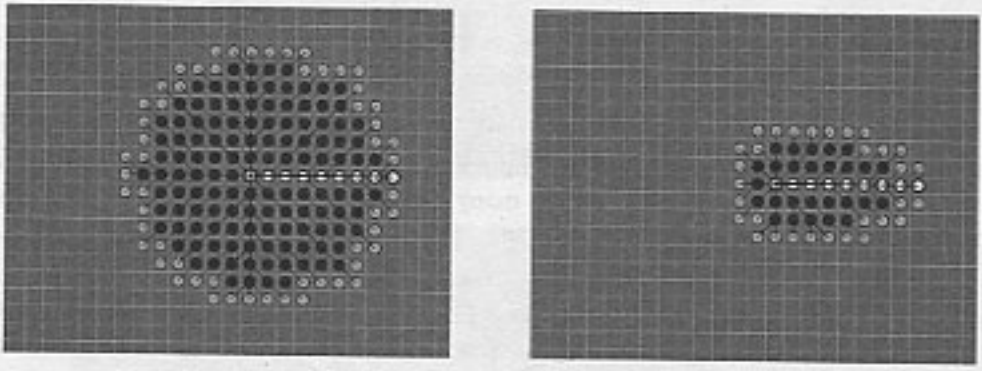

FIGURE 3.3.3. A* search in costly terrain. a: With a heuristic weight of 1. *b:* With a heuristic weight of 5.

guarantees that the search figures out both the closest goal and the best route to it simultaneously.

Weaknesses of A*

Although A* is about as good a search algorithm as you can find, it must be used wisely; otherwise, it can be wasteful of resources. On a large map, hundreds or even thousands of nodes might be in the **Open** and **Closed** lists, which can take up more memory than is available on systems with constrained memory, such as console systems. On any system, A* can take too much CPU time to be affordable.

The case in which A* is most inefficient is in determining that no path is possible between the start and goal locations; in that case, it examines every possible location

accessible from the start before determining that the goal is not among them, as shown in Figure 3.3.4. The best way to avoid this problem is to do a pre-analysis of the map, manually or algorithmically, so that the program can look up whether two locations are accessible from each other—say, on the same island. If they are not, the search is not even attempted.

Further Work

There is much more detail that we could cover, because there are many different path-planning and path-following situations. With an understanding of the workings of A*, one can often figure out how to adapt it to their needs. Take a look at the other articles in this volume for further discussion on the ways to partition a floor space for use in A*, as well as efficiency and aesthetic considerations.

References

Websites

[Woodcock] Woodcock, Steven, "The Game AI Page: Building Artificial Intelligence into Games," available online at www.gameai.com. The "Resources and Links" subpage of this site has many links to Websites that discuss A*, some of which have sample code.

General AI Texts

The following are two recent, very good general AI textbooks, both of which happen to use the agent-centered paradigm for discussing AI:

[Russell95] Russell, Stuart, and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995. Perhaps the best current AI text, it has a couple chapters on search techniques, including A*.

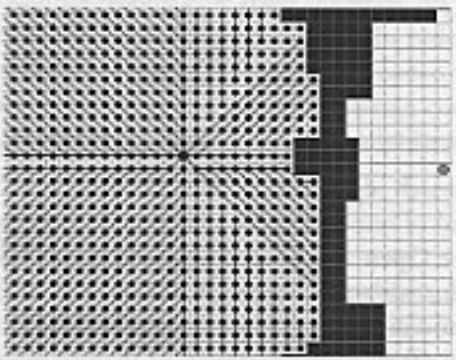

FIGURE 3.3.4. Search in a situation in which there is no path to the goal.

[Nilsson98] Nilsson, Nils J., *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998. Since Nilsson was one of the developers of A*, his discussion of it is valuable if one wants to understand the theory behind it. This text presents formal proofs of properties of A*.

Search Texts

These books discuss the general issues of search, which date back to the early days of AI research:

[Barr81] Barr, Avron, and Feigenbaum, Edward A., eds., *The Handbook of Artificial Intelligence*, volume 1, Addison-Wesley, 1981. A good multivolume survey of major AI issues and important AI programs. This volume includes the discussion of search, including A*.

[Kanal88] Kanal, L., and Kumar, V., eds., *Search in Artificial Intelligence*, Springer-Verlag, 1988. A collection of good articles for those who want to get into the advanced considerations of search, including variations on A* (with imaginative names like B, C, and D!).

[Pearl84] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984. This is perhaps the most complete reference on search algorithms and is referenced by practically everyone else.

[Shapiro] Shapiro, Stuart C., and Eckroth, David, eds., *Encyclopedia of Artificial Intelligence*, 2 volumes, John Wiley & Sons, 1987. A truly excellent collection of articles about most aspects of AI research. Good articles pertinent to this article are "Search," "A* Algorithm," and "Path Planning and Obstacle Avoidance."

3.4

A* Aesthetic Optimizations

Steve Rabin

Computing a path for a character is more than merely an exercise in search algorithms. It also involves creating an aesthetically pleasing path and resulting execution. Computed paths for characters can be improved in three main ways: making the path straighter, making it smoother, and making it more direct. The execution of the path can be improved by simply maximizing responsiveness. All these optimizations result in an experience that is more aesthetically pleasing to the player. Since providing a satisfying experience is the ultimate goal, these things are fairly important and directly impact the code within and surrounding A*.

Straight Paths

Paths calculated by A* often look like they were constructed by someone who was drunk. They weave and bob their way efficiently to the goal, but it sure doesn't look natural. This is a serious problem that undermines the believability of any game's AI. There are two ways to deal with this issue. The first is to promote straight paths within the A* algorithm; the second is to clean up the mess after the path has been calculated.

Promoting straighter paths involves careful cost weighting within the A* algorithm. Consider the two paths computed by A* that are shown in Figures 3.4.1 and 3.4.2.

The amazing observation is that both paths travel the *exact* same distance. Since both paths have identical costs, A* is unable to differentiate between them and simply chooses the first path it stumbles upon. The trick that will make A* choose the straight path is held within the cost function. Simply factor in an extra cost (penalty) if the new step being considered is not straight with the last step. Note that we are not looking at the overall straightness of the path, just penalizing new considerations that are not in line with the last step.

A reasonable penalty is half the normal cost to step in a given direction. The truth is that on a regular grid, any penalty at all (0.0000001) for non-straight choices causes A* to choose the straightest one. However, this is not the case on an arbitrary network.

FIGURE 3.4.1. Typical A* path.

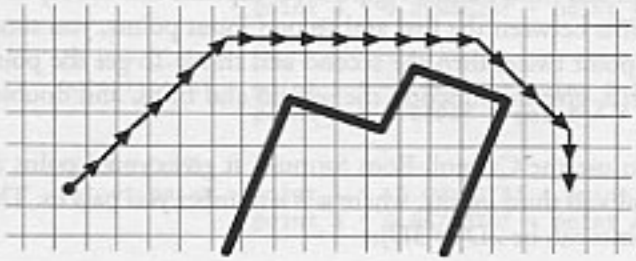

FIGURE 3.4.2. Straightened A* path.

A word of caution: Penalizing non-straight paths results in more work being done by the pathfinder, thus slowing the computation. Obviously, the algorithm has to consider many more permutations in order to find the straightest one. In fact, searches can take significantly more time. However, if hierarchical methods are used, the extra time might not be an issue. Make sure you understand the tradeoffs.

Straight Paths in a Polygonal Search Space

With a polygonal search space, this trick is not very useful. Because triangles aren't uniformly spaced, as a rectangular grid is, similar paths that cost the same are quite rare. Therefore, there is no need to find the straighter path. Instead, there's a different problem: Since triangles can vary greatly in size and proportion, paths are more crooked than ever. The trick is to optimize for straightness after the path has been calculated. Greg Snook's path-finding article, "Simplified 3D Movement and Pathfinding Using Navigation Meshes," discusses an excellent way to handle this problem.

Smooth Paths

Unfortunately, paths computed by A* are usually riddled with sharp turns. Even if you employ a technique to make straighter paths, sharp turns still have the potential to make your characters look like robots. By applying rotational dampening to your

turns, you can probably mask them a little, but you'll swing wide on every sharp corner. There's a much better way.

Straight from the field of computer graphics, there's an algorithm that makes your paths (simply series of points in space) smooth for you. A simple Catmull-Rom spline does the trick because it creates a curve that nails all the control points in the original path (unlike a Bézier curve, which is smoother but doesn't go through the control points). Obviously, it's better to go directly through your points because A* deemed them clear and free of obstacles.

But how do you actually input a list of points and get a smoother list back? The Catmull-Rom formula requires four input points and gives you back a smooth curve between the second and third points. Figure 3.4.3 explains this concept a little better.

To get the points between the first and second input points, you simply give the function the first point twice, then the second and third. To get the points between the third and fourth, give the function the second and third, and double up on the fourth.

Each time you use the Catmull-Rom formula, it gives you a point roughly u% between the second and third inputs, where u is a number you pass in. The following is the formula (points can be 2D or 3D):

```
output_point = point_1 * (-0.5f*u*u*u + u*u - 0.5f*u) +
               point_2 * (1.5f*u*u*u + -2.5f*u*u + 1.0f) +
               point_3 * (-1.5f*u*u*u + 2.0f*u*u + 0.5f*u) +
               point_4 * (0.5f*u*u*u - 0.5f*u*u);
```

Note that if u is zero, the formula gives you point_2. When u is 1, it gives you point_3. As you can see, the spline really does go directly through the input points.

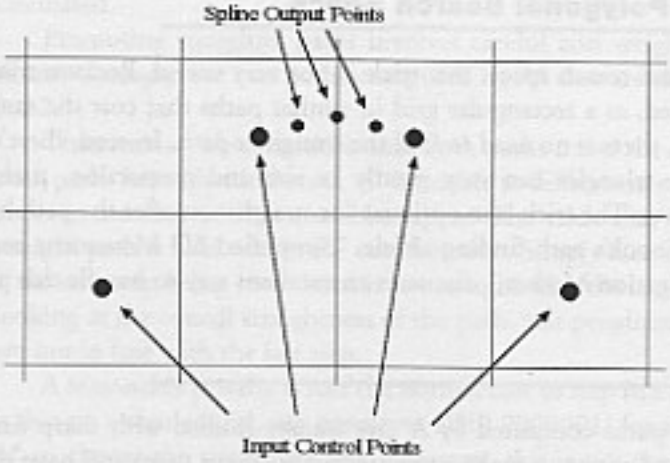

FIGURE 3.4.3. Getting spline points from control points.

A Pre-Computed Catmull-Rom Formula

Since speed is an issue, you might want to dictate that you want u only at certain intervals, such as 0.0, 0.25, 0.5, and 0.75. By freezing all instances of u, you can pre-compute the formula at each u. Note that the formulas can take either 2D or 3D points. The following are some formulas at various intervals:

```
// u = 0.0
output_point = point_2;

// u = 0.25
output_point = point_1 * -0.0703125f + point_2 * 0.8671875f +
               point_3 * 0.2265625f + point_4 * -0.0234375f;

// u = 0.5
output_point = point_1 * -0.0625f + point_2 * 0.5625f +
               point_3 * 0.5625f + point_4 * -0.0625f;

// u = 0.75
output_point = point_1 * -0.0234375f + point_2 * 0.226563f +
               point_3 * 0.8671875f + point_4 * -0.0703125f;

// u = 1.0
output_point = point_3;
```

Equipped with the Catmull-Rom formula, all you need to do is walk through the path that A* found and create a new path. Remember to double up on the first input point when you start, and double up on the last input point when you get to the end. If the A* path has only two points to begin with, simply don't apply the spline to the path.

Since you have a new path with four times the number of points, you might want to look into getting rid of redundant points. Running the new path through a function that prunes co-linear points should dramatically reduce your list.

Figure 3.4.4 shows a typical path before and after the Catmull-Rom spline has been applied. Notice how it's still just a series of points (piece-wise linear), but the path is now much smoother. In the large scale of things, this path is perfectly smooth.

FIGURE 3.4.4. Path points before and after a spline is applied.

Improving the Directness of Hierarchical Paths

A very important A* technique is *hierarchical pathing*. However, the problem is that the resulting paths can be less than ideal. In hierarchical pathing, you pathfind in two distinct steps. You first find the large-scale path, and then you pathfind at the local level. For example, a castle might be broken up into rooms. You might want to get from the dungeon to the throne room. The idea is that you first find the large-scale path between the rooms (by running A* on the connectivity graph of the rooms). Once that path is found, you can then path-find between each connecting room as you encounter it. The result is a huge savings of time. Unfortunately, the overall path can look rather bad because the goal point will always be the door to the next room, thus causing the character to always travel through the center of each door. When doors are arbitrarily large, this can look rather bad. Figure 3.4.5 illustrates the problem.

There is a simple, elegant way to get the ideal path, but it takes roughly twice the computation. The trick is to always path-find to the door beyond the next door. Then, whenever the character crosses through the first doorway, throw away the rest of the path and repeat the process. While the second half of the path is always wasted, it really does create the most direct and aesthetically pleasing route. Figure 3.4.6 shows the final path.

This technique always finds the optimum passage through the doorway because it takes into account the future path. The following is a step-by-step sample path execution guide to show how this method works:

1. Find the best room-to-room path using A* on the connectivity graph of the rooms.
2. The result is to travel the following sequence of rooms: 1, 2, 3, 4.

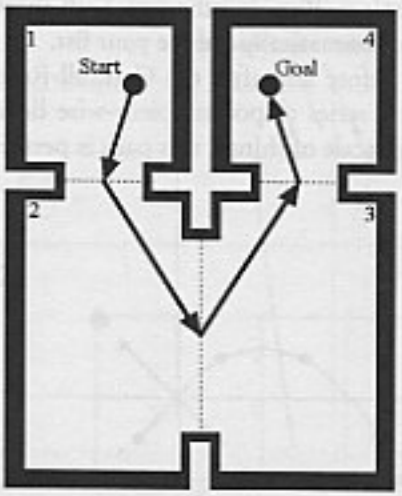

FIGURE 3.4.5. The path through several rooms.

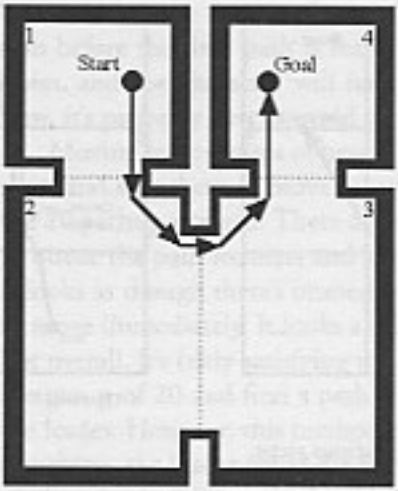

FIGURE 3.4.6. The optimized path through several rooms.

3. Pathfind from Start to subgoal 1 (Figure 3.4.7a).
4. Let the character walk until he enters Room 2.
5. Throw away the remaining path and pathfind to subgoal 2 (Figure 3.4.7b).
6. Let the character walk until he enters Room 3.
7. Throw away the remaining path and pathfind to the final goal (Figure 3.4.7c).
8. Let the character walk to the final goal.

Hierarchical Pathfinding on Open Fields

There is no real difference between a set of connected rooms and a set of connected fields. The same principles apply. The resulting path is not completely straight, but it

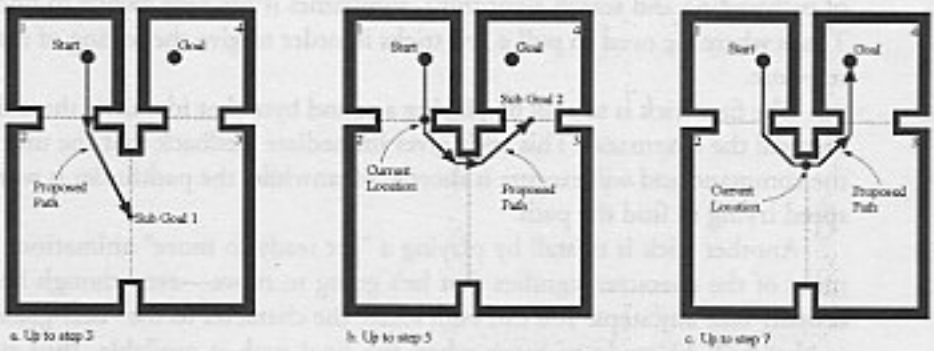

FIGURE 3.4.7. The computed path during various steps.

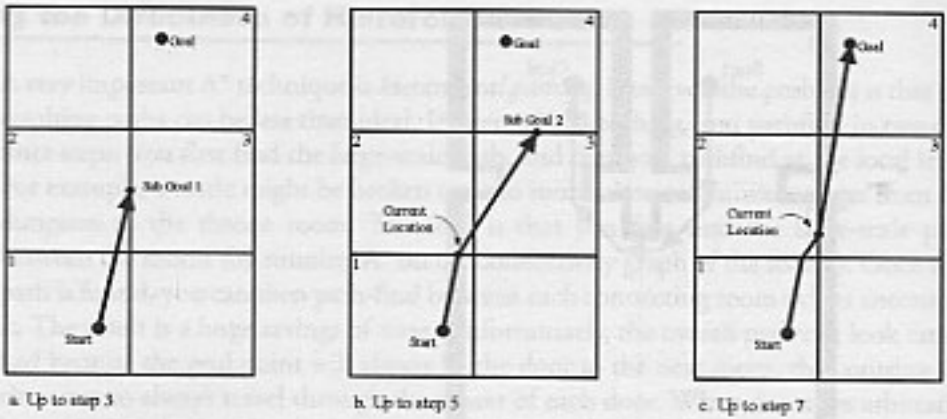

FIGURE 3.4.8. The computed path during various steps.

comes pretty close. You also need to realize that these fields are fairly large relative to the player. Figure 3.4.8 shows the hierarchical steps applied to open fields.

Eliminating Pauses During Hierarchical Searches

Note that every time the character enters a new room, a new local path must be computed. Since this obviously takes time, the character appears to pause as he enters each doorway. The search itself can't be sped up, but the new path request can be anticipated and computed slightly before it's needed. This simple trick keeps the character motion fluid throughout the path.

Maximizing Responsiveness

Controller responsiveness is critical to game play. When a player issues a move command in an RTS, they expect the unit to respond immediately. However, in the world of pathfinding and search algorithms, sometimes it can take awhile to find that path. That's where we need to pull a few tricks in order to give the feeling of instantaneous response.

The first trick is to stall by playing a sound byte that identifies the unit as having received the command. This trick gives immediate feedback that the unit is aware of the command and will execute it shortly. Meanwhile, the pathfinder is working at full speed trying to find the path.

Another trick is to stall by playing a "get ready to move" animation. The movement of the character signifies that he's going to move—even though he might not actually take any steps. You can even rotate the character to the "best-guess" direction so that he'll be ready to move when the final path is available, thus stalling even longer.

You can take the idea further by moving the character in the "best-guess" direction before the final path is ready. Unfortunately, you could be dead wrong in your guess, and the character will have to backtrack. Unless the pathfinder is extremely slow, it's probably best to avoid using this method.

Moving large groups of people at once can take even longer. If the player grabs 20 units and asks them to move across the map, you could be waiting a long time to get the 20 paths processed. There are two tricks to dealing with this situation. The first is to queue the path requests and let each unit move as its request is serviced. This way, it looks as though there's immediate feedback, because at least some of the units start to move immediately. It looks a bit like popcorn popping as each unit starts to move, but overall, it's fairly satisfying to the player. The second trick is to choose a leader in the group of 20 and find a path for only him. Then tell the other 19 units to follow the leader. However, this method can get complicated because there could be massive bunching, the leader could die halfway through the path, and each unit should eventually stop at a unique destination.

Conclusion

All these techniques are designed to make pathfinding more transparent to the player. The goal has always been to find good, direct paths instantaneously. Since that's a tough problem, hopefully you can apply these gems to your current pathfinder in order to get better-looking paths that ultimately feel better to the player.

References

[Patel99] Patel, Amit J., "Amit's Thoughts on Pathfinding," available online at http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Stout96] Stout, Bryan W., "Smart Moves: Intelligent Path-Finding," *Game Developer*, October/November 1996, pp. 28–35, also available online at www.gamasutra.com/features/19990212/sm_01.htm.

3.5

A* Speed Optimizations

Steve Rabin

Traditionally, A* is a slow algorithm that never runs as fast as you'd like. Since there's a long list of optimizations that you could make, its very important to understand why it's slow so you can wisely use your optimization time.

The first thing to notice about A* is that it is at the mercy of the search space. Usually, the sheer number of connections to search is a good indication of how fast A* works. In a rectangular grid of 1,000 by 1,000 squares, there are 1 million possible squares to search. To find an arbitrary path in that kind of world simply takes a lot of work, no matter how optimized your code. The solution is to optimize the search space.

Once the search space has been optimized, it's time to look deeper into the actual A* implementation. Since A* churns through a lot of memory, it's critical to optimize memory allocation as well as each of the data accesses. A* also demands a lot of sorting, but this can be dealt with quickly and efficiently using some specialized data structures.

Lastly, the best way to speed up A* is by not using it at all for simple cases. Construct some kind of test to determine whether you absolutely need to fire up the pathfinder. Many times, simple routes can be determined without using the full-blown A* implementation. For example, try running a blind straight-line path to the goal, testing to see if it collides with walls or other objects. Undoubtedly, there will be times when this simple solution works amazingly well.

Search Space Optimization

Simplifying the Search Space

The biggest win always comes from searching through less data. If you can represent your world as a simplified connectivity graph, A* will work all the faster. Practically, there are several options to choose from. Since speed isn't the only consideration, some other pros and cons are also discussed here. A simple diagram of each technique is provided in Figure 3.5.1.

Rectangular or Hexagonal Grid

Description

A uniform rectangular or hexagonal grid is overlaid onto the world. The size of each grid space is proportional to the size of the smallest character. Therefore, a character in a grid space blocks that space during the A* search. See Figure 3.5.1a.

Pros

- Obstacles and characters can be easily marked in the grid allowing for avoidance. This creates a one-step solution to finding a path through static and dynamic objects.
- Works well for 2D tile-based worlds.

Cons

- Typically results in the largest search space.
- Rectangular grids don't map very well onto 3D worlds.
- Paths tend to look like moves on a chessboard.

Actual Polygonal Floor

Description

In a 3D game world, the floor polygons are specifically marked and used directly as the search space. This polygonal floor is identical to the rendered geometry, thus being arbitrarily simple or complex. See Figure 3.5.1b.

Pros

- Data structure already exists in the 3D world.
- Can be walked through quickly with a BSP tree.

Cons

- Three-dimensional worlds can have arbitrarily high numbers of polygons on the floor.
- Can't represent obstacles such as tables or chairs (because the floor exists beneath these objects).
- Requires algorithmic solution for choosing path points within a polygon.

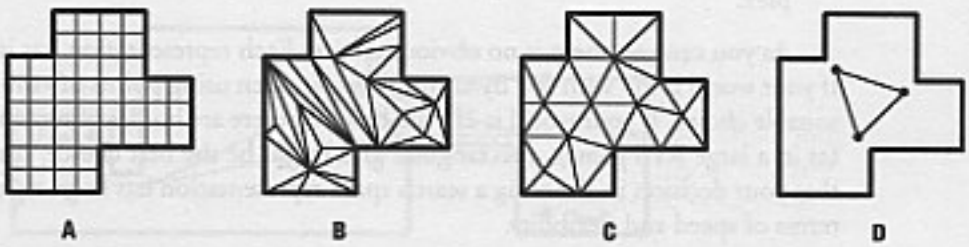

FIGURE 3.5.1. Four options for representing the search space.

Polygonal Floor Representation

Description

An artist or level designer creates a polygonal floor representation that is used exclusively for pathfinding. The polygons can be eliminated in places where characters are not allowed to walk, such as under tables or chairs. See Figure 3.5.1c.

Pros

- Small search space representation.
- Can be walked through quickly with a BSP tree.
- Obstacles can be incorporated in the representation.

Cons

- Requires artist or level designer to construct.
- Can't represent characters within the space.
- Requires algorithmic solution for choosing path points within a polygon.

Points of Visibility

Description

Points are placed at convex corners in the world (sticking out a little from each corner). Each point is then connected to all other points that it can "see". This creates a connectivity graph that describes the minimal paths required to get around walls. See Figure 3.5.1d.

Pros

- Creates minimal search space representation.
- Obstacles can be incorporated in the representation.
- Resulting paths are perfectly direct.

Cons

- Requires algorithmic or designer assistance to create the graph.
- Obstacles can't be removed from the graph if they should be destroyed.
- Can't represent characters within the space.
- Doesn't work well with entities that have large widths, such as a wide formation of characters.
- Worlds with curved walls could cause the graph to become unnecessarily complex.

As you can see, there is no obvious choice. Each representation has its trade-offs. If your world is 3D with few dynamic obstacles, then using points of visibility is a reasonable choice. If your world is 2D tile-based or there are hordes of moving characters (as in a large RTS game), a rectangular grid might be the best choice. Just remember that your decision in choosing a search space representation has huge repercussions in terms of speed and flexibility.

Points of Visibility Explained

Since using points of visibility is an extremely viable option, it's worth explaining a little better. The technique requires that you build up a graph that can be used to get around the world. Points are placed at convex corners and connected to all other points they can see. It's as though a freeway system has been constructed for the sole purpose of getting around walls. The problem now is how you get on and off the freeway.

To get on the freeway, you test the visibility between the starting point and every point on the freeway. Since you can potentially compare thousands of points, it's important that you use other space-partitioning techniques (such as hierarchical pathfinding). Once you have a list of potential on-ramp points, you put them on the A* Open list and begin running the algorithm. With each point you explore, you must test its visibility with the goal point. If you find a point that can see the goal point, you have a potential off-ramp. Figure 3.5.2 shows a simple example.

Hierarchical Pathfinding

Hierarchical pathfinding is an extremely powerful technique that speeds up the pathfinding process. Regardless of which search space representation is used, this technique in effect simplifies that space. Therefore, if your world representation is large, there's still hope. The key is to break up the world hierarchically.

Consider a castle. It can be thought of as a single, large building or as a collection of rooms connected by doors (a large-scale connectivity graph). The pathfinder works in two distinct steps. It first finds the room-to-room path, knowing the starting and ending room. Once that room-to-room path is known, the pathfinder then works on the micro problem of getting from the current room to the next room on the list. Thus, the pathfinder doesn't need to compute the entire path before it takes the first step. The micro path is figured out on a need-to-know basis as each new room is entered. This method significantly cuts down on the search space and the resulting time to compute the path.

This technique really shines if your world is already constructed hierarchically. Even a 3D world could be constructed using a simple building-block paradigm. Consider a circular staircase. Normally, a circular staircase causes most pathfinders a lot of

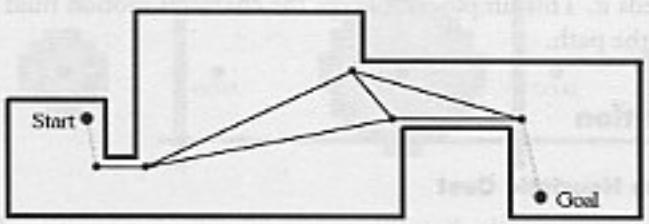

FIGURE 3.5.2. Points-of-visibility example.

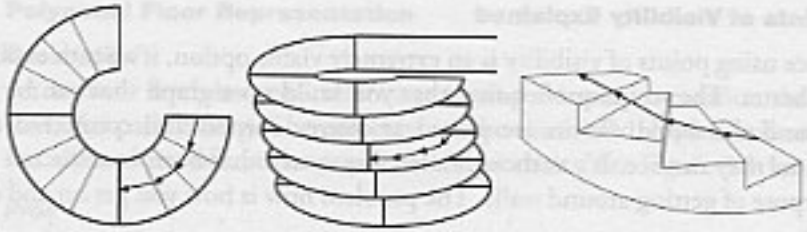

FIGURE 3.5.3. Hierarchical pathfinding on a circular staircase.

grief. The structure is very 3D, mostly circular, and could spiral for a very long distance. A spiral staircase could be built using a quarter-turn piece of the staircase. This piece could then be duplicated indefinitely to create a very tall spiral staircase.

A hierarchical pathfinder could blazingly compute a path up the staircase if it were constructed in this fashion. The pathfinder would first compute the large-scale route through each connecting quarter-turn piece and then would quickly find the local path from the start to the end of each quarter-turn piece. All of a sudden, a complicated path over some rather complicated geometry becomes trivial to compute. Figure 3.5.3 shows an example.

Hierarchical pathfinding isn't restricted to rooms with doors. You can easily extrapolate the idea to huge fields of landscape that are stitched together. Although it's true that there isn't one easily identifiable "door" spot anymore, the entire seam from one field to another becomes the door.

Imagine an immense world created with these stitched pieces of land. Now imagine telling a character to walk from one end to the other. No problem! The pathfinder first finds a route through the network of land pieces and then finds the local path from the current land piece to the next. With a little work, you can even imagine planning a route from the throne room of one castle to the ninth-level dungeon below a completely different castle—even though the castles might be 10 miles apart!

Avoiding Pauses While Computing Local Paths

Since a path is computed every time the character enters a new room, it's important that the character not pause at each door while his new path is constructed. In order to avoid a pause, the path request must be anticipated and completed before the character actually needs it. This simple trick keeps the character motion fluid throughout the execution of the path.

Algorithmic Optimization

Playing with the Heuristic Cost

Designing an algorithm for the heuristic cost can at times seem more like voodoo than science. The idea behind the heuristic cost is to estimate the true cost from a par-

ticular node to the goal. Here's an interesting fact: If you always knew the true cost to the goal, A* would beeline a path to the goal without wasting any search time going down the wrong path. But if the heuristic estimate happens to overestimate the true cost, the heuristic becomes "inadmissible," and the algorithm might not find the optimal path (and might find a terrible path).

The way to guarantee that the cost is never overestimated is by calculating the geometric distance between the node and the goal. When coding A* for the first time, this is the best thing to do until it's time to optimize. Since the cost will never be more than this distance, the optimal path will always be found.

Overestimating the Heuristic Cost

Interesting fact #2: Using a heuristic that routinely overestimates by a little usually results in faster searches with reasonable paths. However, how much should the cost be overestimated? To answer this question, you need to understand what happens when this remaining path cost is artificially bloated.

If the heuristic part of the total cost (*total cost = cost to node + heuristic cost*) is bigger than it should be, it distorts the reasoning by which nodes on the Open list are picked off. Since A* always picks the node with the least total cost, this distortion promotes nodes closer to the goal to be picked.

When you look at an A* search that's trying to find its way around a wall, you can see a shape that develops from the nodes explored (nodes on the Closed list). This shape is the easiest way to see the effects of playing with the heuristic estimate.

When the heuristic equals zero, the search evolves as a circle around the starting point. When the heuristic uses the Euclidean distance to the goal, the search looks like an oval, with the start and goal points the foci. When the heuristic is overestimated, the shape changes to be more of a diamond or hexagon, with the start and goal points at the extreme corners of the shape. Figures 3.5.4, 3.5.5, and 3.5.6 show the growth of the search using various heuristic costs while trying to overcome a large obstacle.

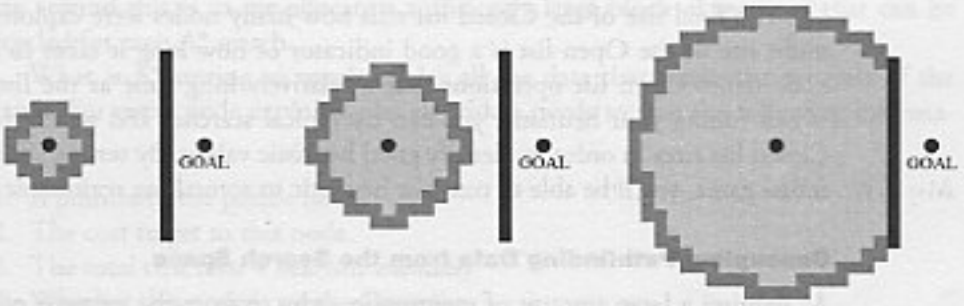

FIGURE 3.5.4. The heuristic cost of zero.

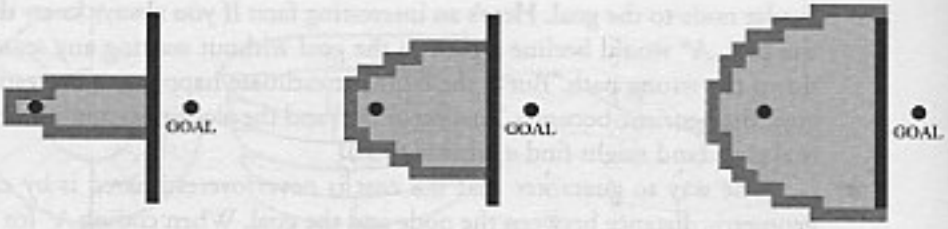

FIGURE 3.5.5. The heuristic cost using Euclidean distance to goal.

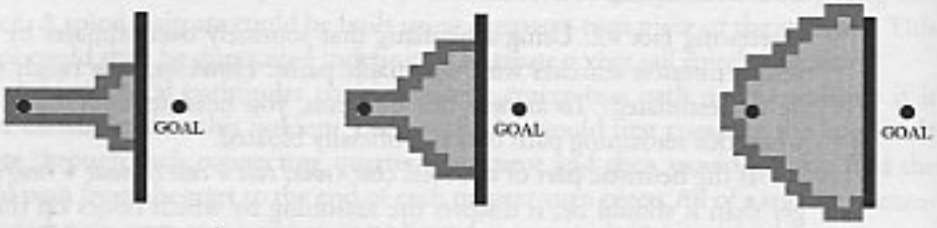

FIGURE 3.5.6. The heuristic cost overestimated.

What does all this mean? It means that by overestimating the heuristic, the search pushes hard on the closest nodes to the goal. This causes pressure for the search to overcome large obstacles that are between the start and goal points of the search. If the actual solution requires backtracking before going to the goal, an overestimating heuristic slows the search. However, if most of the time there's a way to get around large blocking obstacles, the overestimating heuristic is faster. Figure 3.5.7 illustrates this point as the non-overestimating heuristic explores three times more nodes than the overestimating heuristic.

Ultimately, getting the right amount of overestimating requires experimentation. Unfortunately, if the search space is not on a grid, it's probably not possible to accurately observe the shape of the search. Instead, you need to measure indicators such as the size of the Closed list and the maximum size of the Open list.

The final size of the Closed list tells how many nodes were explored; the maximum size of the Open list is a good indicator of how long it takes to explore each node (since Open list operations take a relatively long time as the list grows big). When tuning your heuristic, you can try typical searches and watch the Open and Closed list sizes in order to identify good heuristic values. By testing searches on your actual game, you'll be able to tune the heuristic to something reasonable.

Decoupling Pathfinding Data from the Search Space

A* requires a large amount of memory in order to store the progress of each search. Traditionally, this memory is held inside each searchable node. If the search space is a

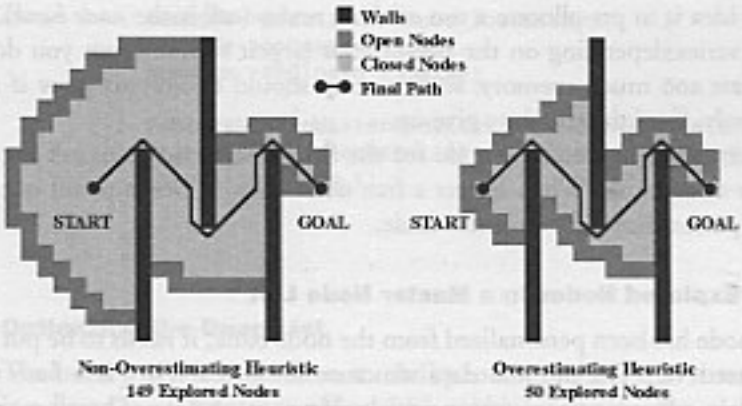

FIGURE 3.5.7. A non-overestimating heuristic vs. an overestimating heuristic.

rectangular grid, each grid square contains pathfinding node data. If the search space is a polygonal mesh, each triangle contains pathfinding node data. Since an individual search almost never covers every single node in the search space, there is no reason to have this incredible amount of memory dedicated to pathfinding. For example, a 1,000 × 1,000 tiled world has 1 million pathfinding nodes just sitting there unused most of the time.

The solution is to decouple the pathfinding node data from the search space. This solution reduces the huge memory overhead and could also speed up searches. Interestingly, by decoupling the node data from the search space, you allow for simultaneous searches, which can now occur because multiple-node data can point to the same real node in the search space. However, it's generally not a good idea to allow simultaneous searches—still, in certain circumstances, it might be useful.

Pre-allocating a Minimum Amount of Memory

Decoupling the node data from the search space requires that each search use some other chunk of memory. We could simply allocate node data on the fly, but A* can churn through hundreds of nodes each search, so this isn't a reasonable solution. A way around this is to pre-allocate a sufficiently large block of memory that can be recycled for each A* search.

What is A* storing so much of? It's all the data that tracks the progress of the search. For every node explored, the algorithm needs to save the following information:

1. A pointer to the parent node
2. The cost to get to this node
3. The total cost (*cost + heuristic estimate*)
4. Whether this node is on the Open list
5. Whether this node is on the Closed list

The idea is to pre-allocate a ton of these nodes (call it the *node bank*). The actual number varies depending on the size of your largest search. Now, you don't want to pre-allocate too much memory, so this array should be able to grow if needed—or alternatively, force the search to give up.

When A* explores a new node for the first time, it needs to ask for a free node from the node bank. When it gets a free node back, it needs to fill out the info in order to personalize it for this new node.

Storing Explored Nodes in a Master Node List

Once a node has been personalized from the node bank, it needs to be put somewhere for fast retrieval. The optimal data structure for this activity is a *hash* table. Hash tables allow constant-time storing and looking up of data. Therefore, we store all explored nodes in this *master node list*. This hash table allows us to instantaneously find out if a particular node is on the Closed list or the Open list. Remember, since the node data memory is already allocated, the hash table contains only pointers to these nodes.

At this point you might ask yourself, "Where is the Closed list?" The answer is that it lives inside the master node list. All explored nodes are stored inside the master node list, and the Closed list just happens to be in the same place. This isn't a problem, because each node is clearly marked as whether it's on the Closed or Open list. So where does the Open list live? The Open list is maintained separately, *but* the master node list also contains pointers to all the same nodes that are on the Open list. Why the duplication? Because sometimes it's faster to find the node you want using the master node list, and sometimes it's faster using the Open list. It's all about speed.

When any given node is explored during the A* algorithm, it's possible that the node was already explored during this same search. To make things simple, you'll want a function that gives you back a pointer to that particular node's data, whether its been searched before or not.

This function first checks the master node list to see if the node has been explored before. If it has, the function simply returns a pointer to that existing node. If the node is not in the master node list, a free node is taken from the node bank, it is initialized to represent the desired node, and its pointer is returned. In effect, the function completely hides the details of allocating new nodes from the node bank and getting nodes that already exist.

```
Node* GetNode( MasterNodeList nodelist, NodeLocation node_location )
{
   //GetNodeFromMasterNodeList accesses the hash table of nodes
   Node* node = GetNodeFromMasterNodeList( nodelist, node_location );
   if( node ) {
      return( node );
   }
   else
   {  //Not in the Master Node List - get new one from the Node Bank
      Node* newNode = GetFreeNodeFromNodeBank();
```

```
        newNode->location = node_location;
        newNode->onOpen = false;
        newNode->onClosed = false;

        //StoreNodeInMasterNodeList places the node into the hash table
        StoreNodeInMasterNodeList( nodelist, newNode );
        return( newNode );
    }
}
```

Optimizing the Open List

The beauty of A* comes from its ability to direct the search toward the most promising directions. The way it achieves this goal is by putting all nodes it could search next into the Open list. It then orders the list from the most promising to the least promising nodes to search. The problem is that the Open list tends to get big, and each time it goes through the A* loop, the most promising node must be extracted from the list. The node to extract is the one with the lowest total cost (*cost to get to the node + heuristic estimate of the remaining cost to the goal*). As it turns out, the best way to store the Open list is to keep it sorted as a priority queue.

A priority queue can be implemented as a binary heap. A *binary heap* is a sorted tree that has the property that the parent always has a lower value than its children. However, there is no ordering among siblings, so a heap is *not* a completely ordered tree. Because of this interesting property, insertions and extractions (removing the lowest element) take only *O(log n)*. Fortunately, that's pretty much all A* needs to do with the Open list.

Implementing a Priority Queue

It's out of the scope of this article to implement a priority queue from scratch, but there's an easy way to implement one using STL. Whether you're using STL or not, be sure to check out a great article about priority queues [Nelson96]. It describes priority queues clearly enough that you could probably construct one without the help of STL. Otherwise, consult any standard algorithm and data structure book for more information.

In order to properly use the priority queue, use the following four operations that A* needs to perform on the Open list:

1. Extract the node with the minimum total cost (and resort the list): *O(log n)*.
2. Insert a new node on the Open list (and resort the list): *O(log n)*.
3. Update the total cost of a node already on the Open list (and resort the list): *O(n+log n)*.
4. Determine whether the Open list is empty: O(1).

STL actually implements a priority queue with something called a *container adapter*. However, the operations that can be performed on it are very limiting. In

fact, it has no interface to perform operation #3 (updating a node's total cost and resorting the list). Therefore, we can't use the STL implementation of a priority queue. However, we can use the STL heap operations on an STL vector container to make our own priority queue!

Listings 3.5.1, 3.5.2, 3.5.3, and 3.5.4 contain the four Open list operations along with the node object, the heap object, and the STL comparison object—all implemented in C++ using STL. In addition, on the CD that accompanies this book, you'll find that Greg Snook's pathfinding article, "Simplified 3D Movement and Pathfinding Using Navigation Meshes," contains almost identical code for implementing an STL priority queue.

A* Using the Optimized Master Node List and Open List

There's nothing too tricky about using the ideas presented, but just in case, the guts of the A* algorithm are implemented in Listing 3.5.5 using the master node list and the Open list. Some other small tricks are also included, such as not searching the node from which the search just came.

Conclusion

Since pathfinding is fundamentally a tough computational problem, the best strategies have always been to simplify the problem. Before any effort is made to optimize the algorithm, ensure that the world is represented in the simplest reasonable way. Once that's decided, it's very important that some kind of hierarchical scheme is also incorporated. Usually this scheme involves some pre-processing that requires level designers and artists to be very involved in the search space representation. Although it adds some overhead to the development of assets, there's no better way to speed up pathfinding.

Once the search space is finalized, it's important to get the pathfinder working correctly in that space before any optimizations are attempted. A* is not a trivial algorithm, and it's extremely difficult to debug if many of the optimizations have been incorporated. When you are ready to optimize, start by decoupling the node memory from the search space. The next step is to implement a priority queue for the Open list and a hash table for the master node/Closed list. Finally, when everything works like a charm and the game is stable, you can play with the heuristic cost. In order to get the best results, you'll want to tune the heuristic several times during development as the game world becomes better defined.

After all these techniques have been implemented, the next step is to cheat. Some techniques for giving the impression of instantaneous pathfinding can be found in the "A* Aesthetic Optimization" article in this book. The trick involves making the player think that a path has been found when in reality you are just stalling. If the player feels that the game is very responsive, the pathfinder appears to be unobtrusive and

transparent, which is the core reason for speeding it up.

Listing 3.5.1: Node Object

```
class Node
{
public:
   NodeLocation location;   // location of node (some location
                            // representation)
   Node* parent;            // parent node (zero pointer represents
                            // starting node)
   float cost;              // cost to get to this node
   float total;             // total cost (cost + heuristic estimate)
   bool onOpen;             // on Open list
   bool onClosed;           // on Closed list
};
```

Listing 3.5.2: Priority Queue Object

```
class PriorityQueue
{
public:
   //Heap implementation using an STL vector
   //Note: the vector is an STL container, but the
   //operations done on the container cause it to
   //be a priority queue organized as a heap
   std::vector<Node*> heap;
};
```

Listing 3.5.3: STL Comparison Function

```
class NodeTotalGreater
{
public:
   //This is required for STL to sort the priority queue
   //(its entered as an argument in the STL heap functions)
   bool operator()( Node * first, Node * second ) const {
      return( first->total > second->total );
   }
};
```

Listing 3.5.4: Four Open List Operations

```
Node* PopPriorityQueue( PriorityQueue& pqueue )
{  //Total time = O(log n)

   //Get the node at the front - it has the lowest total cost
   Node * node = pqueue.heap.front();
```

```
   //pop_heap will move the node at the front to the position N
   //and then sort the heap to make positions 1 through N-1 correct
   //(STL makes no assumptions about your data and doesn't want
   //to change the size of the container.

   std::pop_heap( pqueue.heap.begin(), pqueue.heap.end(),
      NodeTotalGreater() );

   //pop_back() will actually remove the last element from the heap
   //(now the heap is sorted for positions 1 through N)
   pqueue.heap.pop_back();

   return( node );
}

void PushPriorityQueue( PriorityQueue& pqueue, Node* node )
{  //Total time = O(log n)

   //Pushes the node onto the back of the vector (the heap is
   //now unsorted)
   pqueue.heap.push_back( node );

   //Sorts the new element into the heap
   std::push_heap( pqueue.heap.begin(), pqueue.heap.end(),
      NodeTotalGreater() );
}

void UpdateNodeOnPriorityQueue( PriorityQueue& pqueue, Node* node )
{  //Total time = O(n+log n)

   //Loop through the heap and find the node to be updated
   std::vector<Node*>::iterator i;
   for( i=pqueue.heap.begin(); i!=pqueue.heap.end(); i++ )
   {
      if( (*i)->location == node->location )
      {  //Found node - resort from this position in the heap
         //(since its total value was changed before this function
         //was called)
         std::push_heap( pqueue.heap.begin(), i+1,
            NodeTotalGreater() );
         return;
      }
   }
}

bool IsPriorityQueueEmpty( PriorityQueue& pqueue )
{
   //empty() is an STL function that determines if
   //the STL vector has no elements
   return( pqueue.heap.empty() );
}
```

Listing 3.5.5: A* Implemented with a Master Node List and a Priority Queue Open List

```
MasterNodeList g_nodelist;

bool FindPath( GameObject* gameobject, WorldLocation goal )
{
   //Get a path in progress if it exists for this game object with
   //this goal
   //A path may have been started and not finished from last game tick
   //If no path in progress, it returns an empty path structure
   Path* path = GetPathInProgress( gameobject, goal );

   if( !path->initialized )
   {  //The InitializePath function fills out the path structure for
      //this path request
      //It initializes a clean MasterNodeList and a clean Open list
      InitializePath( path, gameobject, goal );

      //Create the very first node and put it on the Open list
      Node* startnode = GetNode( g_nodelist, GetNodeLocation(
         gameobject->pos ) );
      startnode->onOpen = true;        //This node goes on Open list
      startnode->onClosed = false;     //This node not on Closed list
      startnode->parent = 0;           //This node has no parent
      startnode->cost = 0;             //This node has no cost to get to
      startnode->total = GetNodeHeuristic( startnode->location,
         path.goal );
      PushPriorityQueue( path.open, startnode );
   }

   while( !IsPriorityQueueEmpty( path->open ) )
   {
      //Get the best candidate node to search next
      Node* bestnode = PopPriorityQueue( path.open );

      if( AtGoal( bestnode, goal ) )
      {  //Found the goal node - construct a path and exit
         //The complete path will be stored inside the game object
         ConstructPathToGoal( gameobject, path );
         return( true );   //return with success
      }

      while( /*loop through all connecting nodes of bestnode*/ )
      {
         Node newnode;
         newnode.location = /*whatever the new location is*/;

         //This avoids searching the node we just came from
         if( bestnode->parent == 0 ||
            bestnode->parent->location != newnode.location )
         {
            newnode.parent = bestnode;
            newnode.cost = bestnode->cost + CostFromNodeToNode(
```

```
                &newnode, bestnode );
            newnode.total = newnode.cost;

            //Get the preallocated node for this location
            //Both newnode and actualnode represent the same node
            //location, but the search at this point may not want
            //to clobber over the data from a more promising route -
            //thus the duplicate nodes for now

            Node* actualnode = GetNode( g_nodelist,
               newnode.location );

            //Note: the following test takes O(1) time (no searching
            //through lists)
            if( !( actualnode->onOpen && newnode.total >
               actualnode->total ) &&
                !( actualnode->onClosed && newnode.total >
               actualnode->total ) )
            {  //This node is very promising
               //Take it off the Open and Closed lists (in theory)
               //and push on Open
               actualnode->onClosed = false;  //effectively removing it
                                              //from Closed
               actualnode->parent = newnode.parent;
               actualnode->cost = newnode.cost;
               actualnode->total = newnode.total;

               if( actualnode->onOpen )
               {  //Since this node is already on the Open list,
                  //update it's position
                  UpdateNodeOnPriorityQueue( path.open, actualnode );
               }
               else
               {  //Put the node on the Open list
                  PushPriorityQueue( path.open, actualnode );
                  actualnode->onOpen = true;
               }
            }
         }
      }

      //Now that we've explored bestnode, put it on the Closed list
      bestnode->onClosed = true;

      //Use some method to determine if we've taken too much time
      //this tick and should abort the search until next tick
      if( ShouldAbortSearch() ) {
         return( false );
      }
   }

   //If we got here, all nodes have been searched without finding
   //the goal
   return( false );
}
```

References

A* Algorithm

[Heyes-Jones98] Heyes-Jones, Justin, "A* Algorithm Tutorial," available online at www.gamedev.net/reference/programming/ai/article690.asp, 1998.

[Patel99] Patel, Amit J., "Amit's Thoughts on Pathfinding,"" available online at http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Stout96] Stout, Bryan W., "Smart Moves: Intelligent Path-Finding," *Game Developer*, –October/November 1996, pp. 28–35, also available online at www.gamasutra.com/features/19990212/sm_01.htm.

Data Structures

[Lewis91] Lewis, Harry R., *Data Structures and Their Algorithms*, HarperCollins Publishers Inc., 1991.

[Nelson96] Nelson, Mark, "Priority Queues and the STL," *Dr. Dobb's Journal*, also available online at www.dogma.net/markn/articles/pq_stl/priority.htm, January 1996.

3.6

Simplified 3D Movement and Pathfinding Using Navigation Meshes

Greg Snook

Getting an object to move from Point A to Point B intelligently has always been a challenge for the game programmer. Doing the same for an object in 3D space is a greater challenge still. In today's world of complex 3D environments, the task can become overwhelming. This article proposes a rather simple method to help overcome these obstacles and get all your objects safely to Point B with the least amount of work: *Cheat*.

Yes, cheat. Rarely do real-time games have the time to compute true 3D object-to-scene interaction and pathfinding, and the code complexity to do so is often unnecessary for most applications. We are here to find the easier way out. We seek a simple, extendable method to roll our dice and move our mice in a way that looks believable to the player. Let's face it: The easiest ways almost always involve cheating.

In a Nutshell

What we need is a way to simplify 3D space into more familiar 2D terms. Objects in 2D space move in a highly predicable fashion and can be controlled very intuitively by the player. In addition, there is a myriad of 2D search algorithms at our disposal to create intelligent paths for our objects to move on. What we will create is a method allowing our objects to function in a pseudo-3D environment while providing a full-3D presentation for the player. To do this, we employ a mesh of triangular polygons to represent our 3D space as a warped 2D playing field.

The idea stems from the fact that, for most game environments, you can pretty easily predict where objects can and can't move. From that information, a simple set of geometry can be created to define this area as a "walkable" surface area. One way to visualize this area is to imagine a room within a typical 3D environment. Since your characters are humanoid and the planet hosting the game has gravity, you can assume the game objects will spend most of their time on the floor of this room. You can also

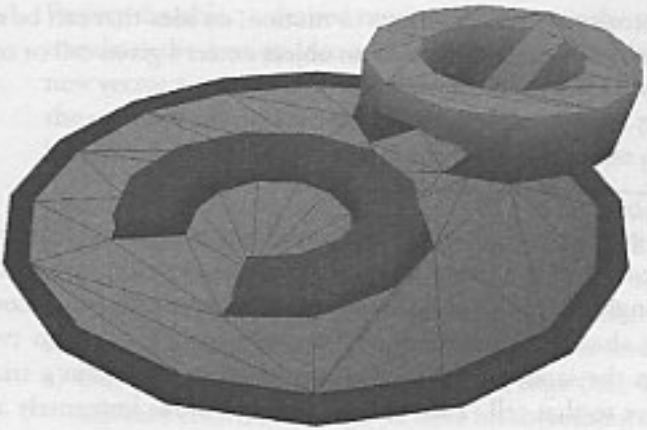

FIGURE 3.6.1. A 3D environment with the associated navigation mesh drawn in wireframe.

assume that they won't be walking through the pillars, desks, soda machines, and other objects sharing the floor space. We can define the remaining portions of the floor with some simple, coarse geometry that covers the open, walkable surface area. Think of this geometry as a sort of polygonal carpet, which we refer to as a *navigation mesh*. It represents the area around which your objects can move within the environment. Figure 3.6.1 shows the 3D environment used in the *navimesh* program available on the CD that accompanies this book. The wireframe polygons show the navigation mesh, which defines the area where objects can move.

In a sense, this navigation mesh object can be treated like the grid systems employed by 2D tile-based games. Each polygon of the mesh can be thought of as a grid cell, except that they attach to adjacent cells on three sides instead of four. With a bit of effort, we can even use this mesh for traditional grid-based algorithms such as line-of-sight detection and pathfinding. The added bonus is that our replacement for the 2D grid can have cells of irregular shape and size, wind up and down stairs and hills, and even overlap itself on things like bridges and catwalks—all while providing access to the same time-tested 2D algorithms we all know and love.

Utilizing a navigation mesh can also reduce the amount of collision testing required between an object and its static environment. Since the navigation mesh already represents an approximation of the open surface area in the environment, our objects need collide with only the mesh edges rather than the true scene geometry. By projecting a control point from the object onto the mesh, we can easily track object movement and collisions with 2D line intersection tests rather than full 3D polygon intersection. In cases in which higher detail is needed for collision, the mesh cells can still link to sets of true scene geometry for refined testing. Objects that collide with a cell edge would then be passed onto routines that resolve the collision with the associated room geometry. Linking process data with the cells in this manner serves as a

quick-and-dirty proximity test for objects in motion, an idea that can be extended to trigger traps, doors, or switches whenever an object enters a given cell or collides with a particular edge.

Construction

Navigation mesh geometry needs to adhere to a few simple rules in order to work correctly. First, it needs to be composed completely of triangles to ensure that each cell is contained in a single plane. Second, the entire mesh must be contiguous, with all adjacent triangles sharing two vertices and a single edge. Finally, no two triangles should overlap on the same plane. That is, any given point within a triangular cell should be exclusive to that cell. This will aid our algorithms immensely and provide believable movement for the player.

The navigation mesh is not intended to be visible to the player. We use it only behind the scenes to limit character movement and determine paths. Therefore, it need consist of only the minimal amount of polygons necessary to represent the area in which objects can move. Highly detailed navigation meshes might produce the most accurate results, but their overhead would be a limiting factor for most real-time games. The mesh should be one that contains only the cells necessary to facilitate believable movement to the player, not one that represents every pebble and twig on the ground.

Roll the Dice and Move Your Mice

To begin, we first examine using the navigation mesh to control object movement in a 3D environment. Once we have our objects happily interacting with the mesh, we can extend its use for pathfinding and line of sight. But first things first: We need to get some objects moving around the confines of the mesh geometry.

We attach objects to the mesh using a control point that will be locked to the navigation mesh surface. This control point may never leave the mesh, but it can move about the surface of the mesh at will. Using our polygonal carpet example, imagine a person standing in a room. The control point can be visualized as a marble sitting directly below the person, resting between his or her feet. Wherever the person is moved in the room, the marble rolls along, always maintaining a position on the carpet directly under the person.

All desired object movement is transferred to this control point, which, in turn, gets resolved on the surface of the navigation mesh. The object is then moved relative to the new control point location. In our example, kick the marble and let it ricochet off the walls, then move the person to the new marble location.

The basic procedure is as follows, given that each object maintains a control point on the mesh and we know which cell of the mesh currently contains the control point:

1. Project the object's desired motion vector onto the plane of its current cell. This translates the motion into a 2D vector along the plane of the cell. We'll call this new vector a motion path and represent it as a 2D line segment. The endpoints of the segment are the starting location of the control point and the desired ending location, both translated to 2D space relative to the plane of the cell.

2. Test the motion path against the cell's 2D triangle edges. Due to the nature of triangles, we know that for our path to exit the cell, it must intersect with exactly one side of the triangle. So, we test the 2D line segment of the motion path against the three line segments representing the cell triangle for any possible intersections. There can be only one of three possible results to this test:
   a) Our path intersects with an unshared edge (i.e., an edge not connected to an adjacent cell). This means we have hit something solid. Resolve the collision of the motion path vector and the cell wall, adjusting the motion path to account for any change in direction, and repeat Step 2.
   b) Our path intersects with a shared edge. Move to the adjacent cell and repeat the entire process from Step 1, projecting our current vector to the plane of the new cell and testing against its walls.
   c) The only remaining possibility is that our motion path does not exit the current cell. We have reached the end of our process and found the cell that hosts the object's new resting position. We translate our resulting 2D motion path endpoint back into 3D space to find the true 3D location of the control point and move the object relative to it.

Obviously, for complex navigation meshes, this can be a very cumbersome process. For each cell encountered, we need to project an arbitrary 3D vector onto a 3D plane. From there we translate the resulting vector along with the cell edges to 2D space, where we can perform our line intersection tests. Once finished moving about the mesh, we need to undo the translations and projections to produce our new control-point position in 3D space.

That's quite an effort to undertake in real time, especially if you have many objects to test or your objects plan to travel over many cells in a given frame. For simple environments, however, this could be plausible and allows for the greatest flexibility in navigation mesh geometry. For complex environments, we can still speed up the process considerably with a bit more careful planning (read: cheating) and an additional navigation mesh geometry rule: *To facilitate fast projections, all cell normals must face in the same direction along a predetermined cardinal axis.*

Imagine our room again with the navigation mesh carpet on the floor. All cell normals of the floor point up, so they meet this new requirement of our mesh. That is to say, all cell normals have a positive y value in our environment. Note that we do not require our new navigation mesh to be flat, we simply no longer allow cells whose normals are 90 degrees or more away from our chosen axis.

With this new rule, the projections become incredibly simple. We simply throw out the dimension along the axis we have chosen. In our carpeted room example, projecting points onto the floor is now as simple as throwing their *y* values out the window. In addition, when we have finished the Motion Path processing along the navigation mesh cells, we have a new 2D (x, z) location and the cell that contains it. Using the cell's plane equation, we can solve for y using our (x, z) location and transform ourselves back into 3D space easily. Our new motion-tracking process is reduced to the following:

1. Create a motion path consisting of the control point and the desired location, reduced to 2D points by tossing out their common axis values.
2. Test the 2D motion path against the sides of the cell triangle as before until a cell is found that contains the destination endpoint of the motion path.
3. Using our new (x, z) control point location and the plane equation of the cell it resides in, solve for y and reconvert our control point back into 3D space.

The *navimesh* sample program includes some simple classes that illustrate this process. In the source code, an object called *NavigationCell* is used to represent a single triangular cell, and *NavigationMesh* represents a collection of those cells. Let's first examine *NavigationCell*, since it does most of the work.

NavigationCell defines a single cell of the mesh with the following members:

```
Plane   m_CellPlane;   // A plane containing the cell triangle
vector3 m_Vertex[3];   // the three vertices of this cell
Line2D m_Side[3];  // a 2D line representing each cell wall
NavigationCell* m_Link[3];    // pointers to cells that attach to
                       // this cell on each of its three
                       // sides.  A NULL link denotes a solid
                       // edge.
```

`Vector3`, `Plane` and `Line2D` are pretty straightforward workhorse classes whose source code is also provided. One point of distinction is that `Line2D` is really treated as a ray passing through two points. It has an implied direction, from Endpoint A to Endpoint B. It also tracks a perpendicular "normal" for the 2D line segment. This normal is used to classify points as being either on the line's left or right side. These notions of "left" and "right" are defined as though you were standing on Endpoint A of the line looking toward Endpoint B. As you see in the source code, the ability to classify points in relation to the line is used quite heavily in our motion processing.

The main use of *NavigationCell* is to perform the step in our process where we determine how a path interacts with the walls of a cell. *NavigationCell* contains a member function to classify a 2D line segment to its three cell walls and return a result. This function, `ClassifyPathToCell()`, is the basic building block of navigation mesh use. The return value of this function can be one of the following enumerated types:

```
enum PATH_RESULT
```

```
{
        NO_RELATIONSHIP = 0,   // the path does not cross this cell
        ENDING_CELL,           // the path ends in this cell
        EXITING_CELL           // the path exits this cell
};
```

In the case where `EXITING_CELL` is the result, the cell wall traversed as well as the 2D point of intersection with the wall are provided to the caller. This allows us to compare any 2D path to the cell and determine what type of intersection occurs. When an intersection with a solid edge occurs, we can use the point of intersection to calculate our new direction and retest. Listing 3.6.1 shows the `ClassifyPathToCell()` function in detail.

NavigationMesh uses this function as needed to process our movement as defined in the preceding steps. The *NavigationMesh* member function `ResolveMotionOnMesh()` manages the entire process, testing each cell encountered using `ClassifyPathToCell()`. It takes in a 3D control point, a pointer to the cell it is currently occupying, and the desired location for the control point after movement has occurred. It returns to the caller the true final location of the control point and the cell the new control point will reside in. Listing 3.6.2 details the use of the `ResolveMotionOnMesh()` function.

Getting There Is Half the Fun

Now that we have seen how to use the navigation mesh to control object movement, we can look into other applications for the mesh. The first obvious use is pathfinding. Keep in mind that our mesh consists of linked cells, which share common edges, just like a grid or hex map. Any path-finding algorithms traditionally applied to a grid or hex map should then reasonably translate to our mesh.

As a matter of fact, using polygonal meshes for search algorithms is no new feat. Since path-finding algorithms were designed to work over databases of linked node data, they work quite nicely across sets of linked vertices. As game programmers, we have become too accustomed to seeing these methods applied to grids and hexes, which is only a small subset of the environments within which they can be used.

Using our navigation meshes, we do add one small wrinkle to the path-finding-over-polygons method: We don't use the mesh vertices. Instead, we use the midpoints of each cell wall. Why? Two reasons come to mind, both of which are arguable depending on your game environment. The first is that we are also using our navigation mesh to limit object movement in the environment. If we generate a path along the cell vertices, we are always traveling on the edges of the cells. This is the most costly movement on a navigation mesh, since moving down the edge of a cell exactly means you are constantly colliding with the cell edge, causing a lot of extra, unnecessary intersection tests.

The second reason is purely aesthetic. If we assume our mesh was designed to use the minimal number of polygons, it stands to reason that there are not many vertices

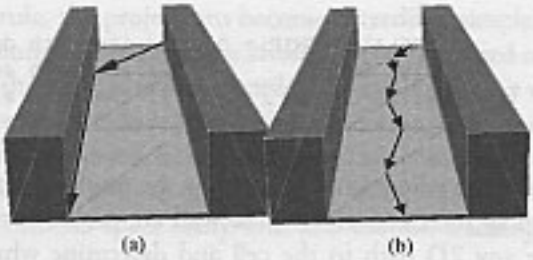

FIGURE 3.6.2. Two overhead views of a sample hallway and navigation mesh showing a path *a:* generated along cell edges and *b:* through cell wall midpoints.

or polygon edges in the body of our open space. Have a look at Figures 3.6.2a and 3.6.2b, which show an overhead view of a hallway, and a reasonable number of polygons to define the open space within it. If we generated a path on the cell edges (Figure 3.6.2a), we would spend most of our time dragging ourselves against the wall of the hallway. Using the cell wall midpoints (Figure 3.6.2b), we can generate a more visually appealing path down the body of the hallway.

As I said, both reasons are arguable. You could simply increase the complexity of your navigation mesh and add code to avoid paths that drag along the solid edges of the mesh, but I have found it easier (and conceptually more intuitive) to move through the wall midpoints. In application, it has also proved to be easier for the level designers creating the mesh to work without having to concern themselves with the placement of extra vertices and cell edges for pathfinding. In essence, all we have really done is offset our mesh vertices to create a more believable path.

So how do we build the path? As in any path-finding situation, you need to choose the best method for your game environment. Best-first searching, Dijkstra's algorithm, and the venerable A* can all be applied to the cells of our navigation mesh. Elsewhere in this book are some excellent explanations of the various search methods, so I will not go into detail on them all here. Check out the articles by Steve Rabin and Bryan Stout in this book for detailed information on A*. You can also check the references at the end of this article for some recommended reading on pathfinding.

For demonstration, the *navimesh* sample code shows how to use A* on the navigation mesh. Although it can be the most complex method to employ, A* can achieve highly accurate results and is often more efficient than other methods in terms of memory use and search time. However, its efficiency hinges on the use of a good heuristic. The heuristic helps steer the search algorithm toward the goal, preventing it from fanning out all over the mesh unless necessary.

The best heuristic to use is purely a game-specific matter. Only you know how well your objects move over your game's terrain, and you need to tune your heuristic accordingly. You might even need to tailor separate heuristics for each object type, taking into account its ability to climb steep grades, corner at high speeds, and so on.

In most cases, however, this heuristic is simply the approximate distance from a given cell to the goal. For the purpose of our demonstration, this is the heuristic we employ.

To run the A* algorithm, we maintain a list of cells that need to be processed. In the *navimesh* sample code, these "Open" cells are held in an ordered list called a *NavigationHeap*. Cells are listed in the order of best to worst cost in terms of the distance required to reach the goal. Therefore, each time we pull a cell off the heap, we know we are dealing with the current "best guess" of the cell that will provide the best path to our goal.

To begin pathfinding, we need to stoke the heap with the first cell, our destination. We then pop and process each cell on the heap until we reach our starting position or run out of heap. If the heap runs dry before we reach the goal, we know there is no path available between our two locations.

To process a cell, examine each of its neighbors. We determine the distance traveled to reach each one by adding the cost associated with our current cell to the distance required to cross the cell to each neighbor. This *Arrival Cost* for each neighbor is then added to the neighbors' own heuristic values to arrive at a priority score for each of the neighboring cells.

We now examine the score, or cost, of each neighboring cell to do one of two things. If the neighboring cell is not currently in the Open heap, we must sort it by its score value. This essentially puts it off for later processing. If the cell is already in the heap, we need to see if our new score is better than the score by which the cell is currently sorted. If the new score is an improvement, we need to move the cell up within the heap to its new priority position for earlier processing. If the new score is not an improvement, we toss it out, since a more optimal path already passes through this cell. In either case, each time a cell is added to the heap or repositioned within the heap, we record the identity of the cell that has set the current *Arrival Cost*.

This is done so that the cells can keep track of the next closest cell to the destination along the generated path. You'll notice that the *navimesh* sample runs the A* algorithm in the reverse search direction, starting at the destination cell and searching for a path backward to our current location. When the search is complete, each cell contains a link to the next cell closest to the goal along the generated path. We can hop through these links in the proper order, from current location to destination, and build a final waypoint list for our game object.

In the sample code, the entire process is run by the `BuildNavigationPath()` member function of the *NavigationMesh* class. It uses the *NavigationHeap* object to maintain a list of *NavigationCells* to be processed. As each cell is pulled from the heap, its `ProcessCell()` function is called, which does the work of testing each neighboring cell, as outlined previously. Cells are added or moved within the heap as necessary, until a path is found. At that point, `BuildNavigationPath()` iterates through the cells on the path, adding their wall midpoints to the final *NavigationPath* waypoint list. Source code for the entire process is shown in Listing 3.6.3.

It Works, But It Ain't Pretty

As you can see by the blue lines drawn in the *navimesh* sample program, building a path through polygonal objects yields a very jagged result. Very rarely will you find that your navigation geometry is set up to produce a straight-line path. The nature of the mesh forces our path to meander from cell to cell, making many abrupt twists and turns (see Figure 3.6.3a). Any object that uses this path verbatim will look very odd indeed to the player. Luckily, we have one final application to discuss that can smooth the path out considerably: line-of-sight determination.

Back when we were working out how to move objects around the mesh, we defined a function, `ClassifyPathToCell()`, to compare a 2D line of motion to a cell. The result of the function told us whether the path ended within the cell, encountered a solid edge, or passed through to an adjacent cell. We can now use that function again to perform a line-of-sight test, smoothing out our path by skipping ahead to the furthest visible waypoint.

Each time we arrive at a waypoint in our path, we look ahead to the next few waypoints in the list. By creating a line of motion from our current position to each of these waypoints, we can quickly test if the waypoints are "visible." To do this, we test the path against each cell between our current position and the waypoint using the `ClassifyPathToCell()` function. If the function returns a solid-wall intersection, we know the waypoint is not visible from our current position. Conversely, if we reach the waypoint without such an intersection, we know the point is visible to us. By searching for the furthest visible waypoint up the chain, we can skip over some of the meandering waypoints and smooth out our path. Figure 3.6.3b shows the new smoother path generated by skipping over the redundant visible waypoints.

This method can be used for all sorts of visibility testing. Using our `ClassifyPathToCell()` function, we can test whether any two points on the mesh can see each other. This has some very useful applications in enemy AI, since you can quickly test whether enemy objects can see the player's position at any given moment. The `LineOfSightTest()` member function of the *NavigationMesh* class details the process for determining point visibility.

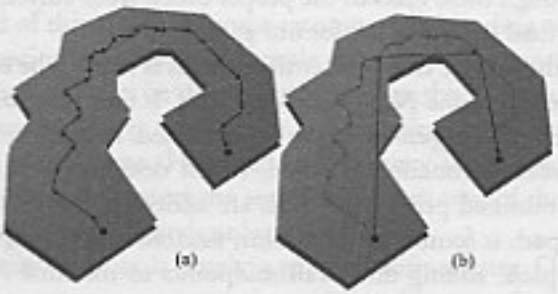

FIGURE 3.6.3. *a:* A sample path generated using A*. *b:* The same path smoothed using line-of-sight testing.

Conclusion

I hope this article has demonstrated that 3D space need not be so computationally complex. My goal was to show that using familiar 2D methods (and a bit of cheating), we can greatly simplify the game environment without impacting the player's 3D experience. The end result is a very flexible and useful tool for moving objects around in 3D space without a boatload of 3D math.

This method allows you to reduce object/scene collision operations, create complex paths, and test for the visibility between any two points in the environment space. However, there is still more that can be done with navigation meshes. They can be constructed of high-order primitives rather than rigid triangles, tessellated like multiresolution meshes for greater path finding detail near the camera, or even animated to represent fluid surfaces. For now, I leave the exploration of these ideas to you. Give the meshes a try and see what new uses you can layer on top of this foundation.

Listing 3.6.1: Intersecting a 2D Line with a Cell of the Navigation Mesh

```
NavigationCell::PATH_RESULT NavigationCell::ClassifyPathToCell(const
Line2D& MotionPath, NavigationCell** pNextCell, CELL_SIDE& Side,
vector2* pPointOfIntersection)const
{
    int InteriorCount = 0;

    // Check our MotionPath against each of the three cell
    //   walls
    for (int i=0; i<3; ++i)
    {
        // Classify the MotionPath endpoints as being either
        // ON_LINE, or to its LEFT_SIDE or RIGHT_SIDE.
        // Since our triangle vertices are in clockwise order,
        // we know that points  to the right of each line are
        // inside the cell.  Points to the left are outside.
        // We do this test using the ClassifyPoint function of
        // Line2D

        // If the destination endpoint of the MotionPath
        // is Not on the right side of this wall...
        if (m_Side[i].ClassifyPoint(MotionPath.EndPointB()) !=
        Line2D::RIGHT_SIDE)
        {
            // ..and the starting endpoint of the MotionPath
            // is Not on the left side of this wall...
            if
            (m_Side[i].ClassifyPoint(MotionPath.EndPointA()) !=
            Line2D::LEFT_SIDE)
            {
                // Check to see if we intersect the wall
                // using the Intersection function of
                // Line2D
```

```
                Line2D::LINE_CLASSIFICATION IntersectResult =
                MotionPath.Intersection(m_Side[i],
                pPointOfIntersection);

                if (IntersectResult ==
                Line2D::SEGMENTS_INTERSECT ||
                IntersectResult ==
                Line2D::A_BISECTS_B)
                {
                    // record the link to the next
                    // adjacent cell (or NULL if no
                    // attachment exists) and the
                    // enumerated ID of the side we hit.

                    *pNextCell = m_Link[i];
                    Side = (CELL_SIDE)i;
                    return (EXITING_CELL);
                }
            }
        }
        else
        {
            // The destination endpoint of the MotionPath
            // is on the right side.  Increment our
            // InteriorCount so we'll know how many walls we
            // were to the right of.

            InteriorCount++;
        }
    }

    // An InteriorCount of 3 means the destination endpoint of
    // the MotionPath was on the right side of all walls in the
    // cell.  That means it is located within this triangle,
    // and this is our ending cell.

    if (InteriorCount == 3)
    {
        return (ENDING_CELL);
    }

    // We reach here only if the MotionPath does not
    // intersect the cell at all.
    return (NO_RELATIONSHIP);
}
```

Listing 3.6.2: Resolving Motion on a Navigation Mesh

```
void NavigationMesh::ResolveMotionOnMesh(const vector3& StartPos,
NavigationCell* StartCell, vector3& EndPos,
NavigationCell** EndCell)
{
    // create a 2D motion path from our Start and End
```

```
// positions, tossing out their Y values to project them
// down to the XZ plane.
Line2D MotionPath(vector2(StartPos.x,StartPos.z),
vector2(EndPos.x,EndPos.z));

// these three will hold the results of our tests against
// the cell walls
NavigationCell::PATH_RESULT Result =
NavigationCell::NO_RELATIONSHIP;
NavigationCell::CELL_SIDE WallNumber;
vector2 PointOfIntersection;
NavigationCell* NextCell;

// TestCell is the cell we are currently examining.
NavigationCell* TestCell = StartCell;

//
// Keep testing until we find our ending cell or stop
// moving due to friction
//
while ((Result != NavigationCell::ENDING_CELL)
        && (MotionPath.EndPointA() !=
        MotionPath.EndPointB()))
{
    // use NavigationCell to determine how our path and
    // cell interact
    Result = TestCell->ClassifyPathToCell(MotionPath,
        &NextCell, WallNumber, &PointOfIntersection);

    // if exiting the cell...
    if (Result == NavigationCell::EXITING_CELL)
    {
        // Set if we are moving to an adjacent cell or
        // we have hit a solid (unlinked) edge
        if(NextCell)
        {
            // moving on. Set our motion origin to the
            // point of intersection with this cell
            // and continue, using the new cell as our
            // test cell.
            MotionPath.SetEndPointA(PointOfIntersection);
            TestCell = NextCell;
        }
        else
        {
            // we have hit a solid wall.
            // Resolve the collision and correct our
            // path.
            MotionPath.SetEndPointA(PointOfIntersection);
            TestCell->ProjectPathOnCellWall(WallNumber,
            MotionPath);

            // add some friction to the new MotionPath
            // since we are scraping against a wall.
            // we do this by reducing the magnitude of
```

```
                // our motion 10%
                vector2 Direction =
                    MotionPath.EndPointB() -
                    MotionPath.EndPointA();
                Direction *= 0.9f;
                MotionPath.SetEndPointB(MotionPath.EndPointA() +
                Direction);
            }
        }
        else if (Result == NavigationCell::NO_RELATIONSHIP)
        {
            // Although theoretically we should never
            // encounter this case, we do sometimes find
            // ourselves directly on a vertex of the cell.

            // This can be viewed by some routines as being
            // outside the cell.  To accommodate this rare
            // case, we force our starting point into the
            // current cell by nudging it back so we may
            // continue.

            vector2 NewOrigin = MotionPath.EndPointA();
            TestCell->ForcePointToCellCollumn(NewOrigin);
            MotionPath.SetEndPointA(NewOrigin);
        }
    }

    // we now have our new host cell
    *EndCell = TestCell;

    // Update the new control point position,
    // solving for Y using the Plane member of the
    // NavigationCell
    EndPos.x = MotionPath.EndPointB().x;
    EndPos.z = MotionPath.EndPointB().y;
    TestCell->MapVectorHeightToCell(EndPos);
}
```

Listing 3.6.3: Building a Navigation Path on the Mesh Using A*

```
bool NavigationMesh::BuildNavigationPath(NavigationPath& NavPath,
NavigationCell* StartCell, const vector3& StartPos,
NavigationCell* EndCell, const vector3& EndPos)
{
    bool FoundPath = false;

    // Increment our path finding session ID
    // This identifies each path finding session
    // so we do not need to clear out old data
    // in the cells from previous sessions.
    ++m_PathSession;

    // load our data into the NavigationHeap object
```

```
// to prepare it for use.
m_NavHeap.Setup(m_PathSession, StartPos);

// We are doing a reverse search, from EndCell to
// StartCell.  Push our EndCell onto the Heap as the first
// cell to be processed.

EndCell->QueryForPath(&m_NavHeap, 0, 0);

// process the heap until empty, or a path is found
while(m_NavHeap.NotEmpty() && !FoundPath)
{
    NavigationNode ThisNode;

    // pop the top cell (the open cell with the lowest
    // cost) off the Heap
    m_NavHeap.GetTop(ThisNode);

    // if this cell is our StartCell, we are done
    if(ThisNode.cell == StartCell)
    {
        FoundPath = true;
    }
    else
    {
        // Process the Cell, Adding its neighbors to the
        // Open Heap as needed
        ThisNode.cell->ProcessCell(&m_NavHeap);
    }
}

// If we found a path, build a waypoint list
// out of the cells on the path
if (FoundPath)
{
    NavigationCell* TestCell = StartCell;
    vector3 NewWayPoint;

    // Setup the Path object, clearing out any old data
    NavPath.Setup(this, StartPos, StartCell, EndPos,
    EndCell);

    // Step through each cell linked by our A* algorithm
    // from StartCell to EndCell
    while (TestCell && TestCell != EndCell)
    {
        // add the link point of the cell as a way point
        // (the exit wall's center)
        int LinkWall = TestCell->ArrivalWall();

        NewWayPoint = TestCell->WallMidpoint(LinkWall);
        NewWayPoint = SnapPointToCell(TestCell,
        NewWayPoint);
        // just to be sure
```

```
            NavPath.AddWayPoint(NewWayPoint, TestCell);

            // and on to the next cell
            TestCell = TestCell->Link(LinkWall);
        }

        // cap the end of the path.
        NavPath.EndPath();
        return(true);
    }

    // no path exists between the two points provided.
    // i.e. "you can't get there from here"
    // This will never happen on a contiguous mesh.
    return(false);
}


bool NavigationCell::ProcessCell(NavigationHeap* pHeap)
{
    if (m_SessionID==pHeap->SessionID())
    {
        // once we have been processed, we are closed
        m_Open  = false;

        // query all our neighbors to see if they need to be
        // added to Open heap
        for (int i=0;i<3;++i)
        {
            if (m_Link[i])
            {
                // The Distances between the wall midpoints
                // of this cell are held in the order
                // ABtoBC, BCtoCA and CAtoAB.

                // abs(i-m_ArrivalWall) is a formula to
                // determine which distance measurement
                // to use.  We add this distance to known
                // m_ArrivalCost to compute the total cost
                // to reach the next adjacent cell.

                m_Link[i]->QueryForPath(pHeap, this,
                m_ArrivalCost+m_WallDistance[abs(
                i-m_ArrivalWall)]);
            }
        }
        return(true);
    }
    return(false);
}

bool NavigationCell::QueryForPath(NavigationHeap* pHeap,
NavigationCell* Caller, float arrivalcost)
{
    if (m_SessionID!=pHeap->SessionID())
```

```
    {
        // this is a new session, reset our internal data
        m_SessionID = pHeap->SessionID();

        if (Caller)
        {
            m_Open  = true;
            ComputeHeuristic(pHeap->Goal());
            m_ArrivalCost = arrivalcost;

            // Remember the triangle wall this caller is
            // entering from
            if (Caller == m_Link[0])
            {
                m_ArrivalWall = 0;
            }
            else if (Caller == m_Link[1])
            {
                m_ArrivalWall = 1;
            }
            else if (Caller == m_Link[2])
            {
                m_ArrivalWall = 2;
            }
        }
        else
        {
            // We are the cell that contains the starting
            // location of the A* search.

            m_Open  = false;
            m_ArrivalCost = 0;
            m_Heuristic = 0;
            m_ArrivalWall = 0;
        }
        // add this cell to the Open heap
        pHeap->AddCell(this);
        return(true);
    }
    else if (m_Open)
    {
        // A true m_Open means we are already in the Open
        // Heap.  If this new caller provides a better path,
        // adjust our data.  Then tell the Heap to resort our
        // position in the list.

        if ((arrivalcost + m_Heuristic) < (m_ArrivalCost + m_Heuristic))
        {
            m_ArrivalCost = arrivalcost;

            // Remember the triangle wall this caller is
            // entering from
            if (Caller == m_Link[0])
            {
                m_ArrivalWall = 0;
```

```
            }
            else if (Caller == m_Link[1])
            {
                m_ArrivalWall = 1;
            }
            else if (Caller == m_Link[2])
            {
                m_ArrivalWall = 2;
            }
            // ask the heap to resort our position in the
            // priority heap
            pHeap->AdjustCell(this);
            return(true);
        }
    }
    // this cell is closed
    return(false);
}

void NavigationCell::ComputeHeuristic(const vector3& Goal)
{
    // our heuristic is the estimated distance (using the
    // longest axis delta) between our cell center point
    // and the goal location


    float XDelta = fabs(Goal.x - m_CenterPoint.x);
    float YDelta = fabs(Goal.y - m_CenterPoint.y);
    float ZDelta = fabs(Goal.z - m_CenterPoint.z);

    m_Heuristic = __max(__max(XDelta,YDelta), ZDelta);
}
```

References

[Patel99] Patel, Amit J., "Amit's Thoughts on Pathfinding," available online at http://theory.stanford.edu/~amitp/GameProgramming/, November 27, 1999.

[Heyes-Jones99] Heyes-Jones, Justin, "A* Algorithm Tutorial," available online at www.gamedev.net/reference/programming/ai/article690.asp, November 27, 1999.

[Stout96] Stout, Bryan W., "Smart Moves: Intelligent Path-Finding," Game Developer, also available online at www.gamasutra.com/features/19990212/sm_01.htm, October 1996.