Warp Speed: Path Planning for *Star Trek®: Armada*

Dr. Ian Lane Davis

Mad Doc Software
186 Elm Street
Andover, MA 01810
Mad_Dr_I@MadDocSoftware.com

Abstract

Star Trek®: Armada is a Real Time Strategy game (RTS) involving space combat between fleets of Federation, Romulan, Klingon, and Borg starships. Each side can have up to approximately 30 capital ships on its side, and maps can vary considerably in aspect ratio and size. Obstacles in space include moving asteroid belts (with gaps through which ships can pass), nebulae which can slow ships down and cripple ships' systems, starbases, black holes, ion storms, other ships, etc

The nature of a 3D RTS is that it must be fast, and the path planning must be considerate with its CPU usage. Core algorithms such as the A* search algorithm are becoming widely accepted in the game industry. However, the key to efficient path planning is how the environment is spatially decomposed, what is left to search-based planning, what is avoided with reactive collision avoidance, and what is simply prevented with collision prevention.

Star Trek®: Armada uses a quadtree decomposition of the playing field which greatly reduces the number of cells that A* needs to search. However, care must be taken to perform some kind of rubber-banding to make the paths look natural. We also use reactive avoidance for most moving objects, and as a final safety measure we actually prevent ships from having overlapping shield ellipsoids in a post-simulation step. In this paper we shall describe the implementation of the path planning system in detail.

The Quadtree

Having decided that we want to find the optimal path from point A to point B, and knowing that our terrain maps are 1) known fully by the computer and 2) mostly static, we can safely decide to use the A* algorithm for path planning [Latombe91] [Tanimoto87]. However, A* is a generic search algorithm which in its purest form simply finds paths through a directional graph with costs on the edges. The key to doing path planning cheaper and better is to choose the cells we search through intelligently.

The cells we search through need to have two principal qualities: 1) each cell must have uniform movement costs within its borders, and 2) each cell must be convex. These qualities guarantee that if we enter the cell, we can move in a straight line across it to any exit point without changing movement costs or being obstructed. As it is most often applied in games, the set of cells underlying the A* is actually a grid of evenly spaced, uniformly sized squares or rectangles (see, for example, *Dark Reign* or *Civilization: Call To Power*). The directional graph is really the array of cells with implicit links between every two adjacent cells, usually 8-connected. Each cell can be assigned a movement cost depending on the speed of traversing the underlying terrain (for more, see [Stout98]).

The only problem with choosing a uniform grid for path planning is that it can be wickedly inefficient. Many strategy games, for instance, can have maps with 256 x 256 cells, which gives us 65536 cells we may need to search through. A naïve implementation can also have one particular worst case: when no path exists from point A to point B, A* will search the whole map[1]. In such games, path planning can use as much as 50% of the CPU. Thus, we have the following rule to follow:

> *The Bigger Is Better Law of Spatial Decomposition for Path Planning (BIBLOSD): Use the largest convex regions you can as the basis for search-based path planning.*

The bigger the regions, the fewer there are to search, and this makes your game faster. Assuming you have some minimum obstacle size, you will have some minimal acceptable cell size, so just making your uniform cells bigger usually doesn't achieve all of your goals (fast planning & good paths). There are a vast number of different convex decompositions that can be used, including some optimal decompositions that will find the smallest number of cells (but which are expensive to compute), and some not-quite optimal decompositions which are a lot faster (for example, see [Latombe91] for an *nlogn* trapezoidal decomposition, see also [Preparata85] for more spatial algorithms). In *Star Trek®: Armada* we have large areas of vast open space and vast areas of slow terrain, and small pockets of impassable terrain, and a quadtree provided a good combination of fast pre-computation (and recomputation) and a great decrease in the number of cells from a uniform grid (with a nice bonus

[1] This effect can be mitigated by preprocessing the map into connected regions of passable terrain. In *Dark Reign*, we had 11 different maps of connected regions (one for each movement modality: infantry, treaded, hover, etc.). Don't even bother planning a path if point A and point B are in different regions.

of being a fairly straightforward implementation compared to many other convex decompositions). See Figure 1, for a screenshot of the quadtree in the game.

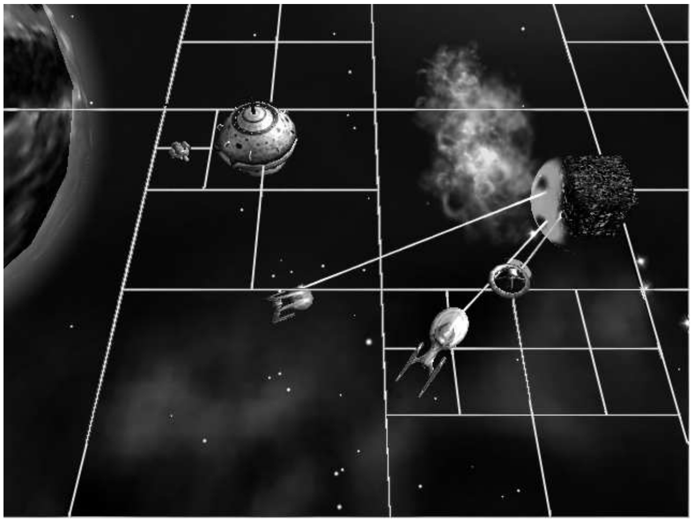

Figure 1: Federation &Borg ships in a game map with quadtree gridlines overlaid.

Preprocessing the map consists of three steps:

1. Add all regions of high terrain cost
2. Simplify the quadtree
3. Extract the graph of neighboring cells

For the quadtree, you start with the map logically considered to be one big empty cell. When you add a region of higher cost, you recursively do the following:

1. If the added region entirely covers the current cell, set the current cell's cost to the new cost.
2. Otherwise, divide the current cell into four equal size quarters, and recursively add the new region to each child cell.

In this fashion, you add all regions of higher cost to the map. In our game, the slow regions are nebulae (five different flavors each with its own speed) and asteroid belts. The impassable regions are asteroid fields, black holes, starbases, and a few other special objects. When they are all added, step 2 has us go through the quadtree and take any divided cell that happens to end up with four children of equal terrain cost, and undivided it (which happens frequently when lots of small similar regions are added in). Note that this step could be integrated into step 1 if desired.

Finally, the graph of neighboring cells needs to be extracted from the quadtree (we use a 4-connected quadtree). This is done recursively, and each step of the recursion has several steps:

1. If the current cell is impassable, skip all remaining steps
2. If the current cell is undivided, add it to list of cells that border the north, south, east, and west of the current cell (trivially true). Skip the remaining steps.
3. Since the current cell is divided, recurse to all four children.
4. Create the list of internal cells that border our north border by merging the north border lists of our northeast and northwest children. Make similar lists for our internal cells that border our east, west, and south edges. These lists are used for making neighbor links outside of our cell.
5. Create neighbor links for our internal cells. There are four internal borders: top & bottom (vertical), and left & right (horizontal). For the right border, create a neighbor link between each cell in the northeast child's south border list and each cell in the southeast child's north border list.

An example of a neighbor graph superimposed on a quadtree is shown in Figure 2. Note that the links are directional, although for simplicity they are not shown as such.[2]

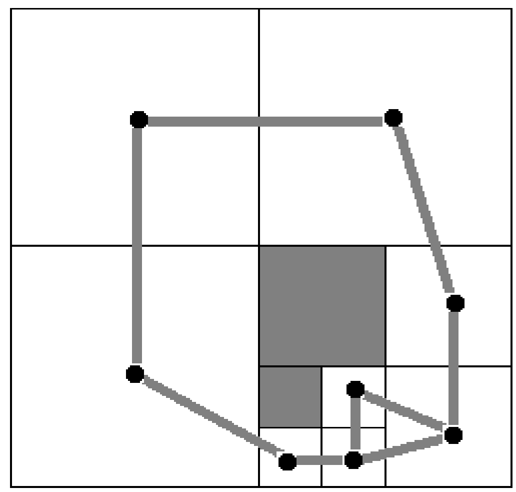

Figure 2. The neighbors graph for a quadtree

Path planning is now a simple matter of performing A* on the graph. However, the paths can be ugly, and can actually take a ship a fair bit out of its way if we just move it through the centers of these cells. In fact, all convex decompositions create path-shape artifacts. Even a uniform grid creates artifacts, as in many games the units/characters always move at 45 degree or 90 degree angles as they pass through 8-connected cells. The one large side effect of the BIBLOSD is that bigger cells have more obvious artifacts; figure 3 shows a path with such artifacts. What we need to do is to consider the results of

[2] In *Star Trek: Armada* we created directional links *out* of impassable cells, but not into them since the physics system and other special effects could sometimes move a ship into what should be impassable region (this is not shown in the diagram)

the A* as a list of which cells to pass through, and perform some sort of rubber banding on the path in order to pull it tight to the edges we want to follow.

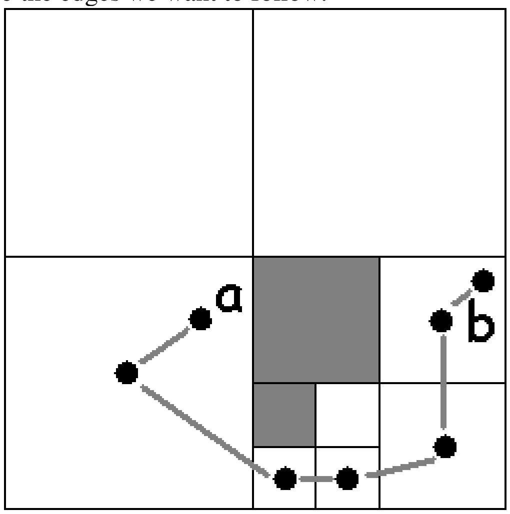

Figure 3. A not entirely good path

Rubber Banding, Step 1

Our solution has two steps: 1) We find the best corners of the cell to go to, and 2) We eliminate unnecessary points on the path. In the first step, we take advantage of the A* code we already have for searching for the best path on a generic directed graph. We create a secondary A* graph which includes as nodes only the start and end points and the corners of the shared edges of each pair of connected cells on our path. Each node then is linked just to the two nodes on the next edge we cross on the path. We then perform a second A* search. Although A* in general can be slow, since we are only connecting the cells forwards, and each cell is only connected to two other cells, the second A* is *linear to the length of the path*. Figure 4 shows a better path that was found with this method.

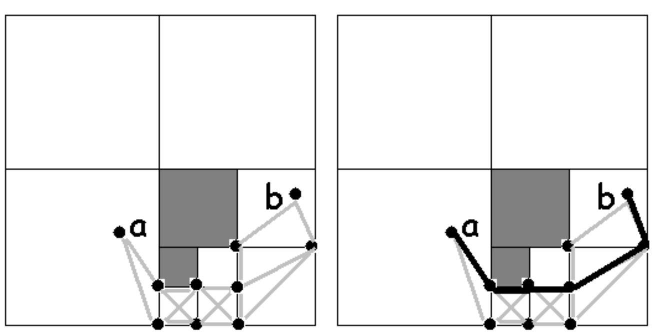

Figure 4. Better, but not quite there. Left shows the forward-linked-only A graph. Right shows path.*

Rubber Banding, Step 2

The final step is to move along the path from start to finish, and maintain the LVP (Last Visible Point) on the path. The LVP is defined as the earliest point on the path to which we can draw a straight line from the current point and not have the line exit any of the cells on our path. If we are at point 6 on the path, and the current LVP is 3, we can safely eliminate points 4 and 5 from the path since we have a valid straight line from 3 to 6. Whenever the LVP changes, we eliminate any points between the old LVP and the last point before the LVP became invalid.

In fact, this operation is also linear in the length of the path. When we move from a point across a border into the next cell, two rays define “Visibility Regions” from the LVP. As we go along our phase 1 path, the Visibility Region from the LVP can only shrink. Once we encounter a point which is not in the Visibility Region, we make the point just before it the new LVP, and we eliminate points between the old and the new LVPs. This gives us our final path.

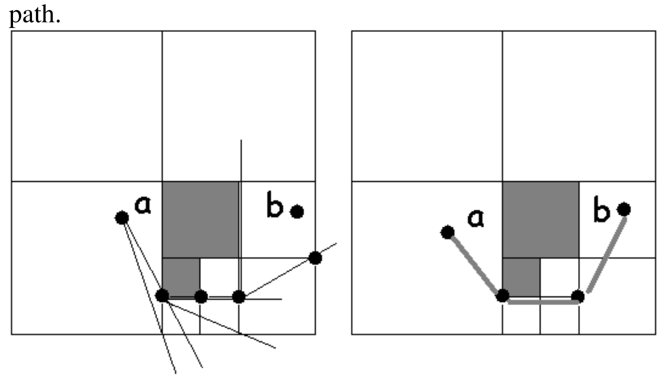

Figure 5. Visibility regions (left) and the final path (right)

Support Systems: Collision Avoidance & Collision Prevention & Misc

We now have a fast system for path planning in *Star Trek®: Armada*. Whenever a ship wants to move, it can find a nice looking path efficiently. However, there are still difficulties, and they are related to moving obstacles. The system as we’ve implemented it does not plan paths around moving obstacles, and in an efficient system like this, the likelihood that two ships will take the same route for part of their respective paths is high. Looking at the path in Figure 5, it is easy to see that if a ship goes from point A to point B while another is moving from anywhere near B to anywhere near A their paths overlap in the lower section.

We rely on collision avoidance and *collision prevention* to keep the ships from smacking into each other. The reactive collision avoidance is similar to systems used for robots. We have two systems that we are evaluating. The first uses potential fields [Latombe91] [Davis96] to apply steering forces for the ships away from objects nearby. The second evaluates a number of different steering arcs [Kelly95] weighting obstacles negatively and the goal (next point on path) positively, and chooses the highest value arc.

The secondary system is called collision prevention, and it has no analog in the real world. To make the ships look intelligent and the starship pilots to appear reasonably trained (and to prevent weapons from being fired within another ship's shields!) we must guarantee that under no circumstances will two ships' shield ellipsoids overlap. Since there are many forces that can act on a ship (black holes, explosions, special weapons, engines, etc.) we simply *do not allow* overlap. To achieve this, on every simulation cycle of the game, we look at each ship and check it against each nearby ship (using geometric hashing, aka buckets, for efficiency [Sedgewick92]). If the shield ellipsoids overlap by any amount, we simply separate the ships by that amount. In practice, this looks fine due to the nice curves of the ellipsoids. As long as we never let them overlap, and separation looks smooth.

Conclusions

Although the basics of path planning are beginning to be understood within the computer game development community, the key to planning paths efficiently is in the proper decomposition of the playing field into convex regions. Additionally, any convex decomposition from uniform grids to optimal decompositions will result in the paths having artifacts that can affect gameplay and reveal too much of the underlying path planning structure to the game player!

For *Star Trek®: Armada* we have chosen a quadtree spatial decomposition and developed a two step rubber banding approach to clean up the paths. In the first mission of *Star Trek®: Armada*, in order to achieve the same planning resolution in a uniform grid as we have in the quadtree, we would have needed 16384 cells. The quadtree, on the other hand, uses only 2191. This makes both the average and worst case path searches with A* significantly faster. Although there are more optimal decompositions, the speed of pre-computation and the ease of maintenance make this decomposition very appealing. Furthermore, there is a linear time (in the length of the path counted in cells traversed) two step clean up of the paths to remove the artifacts of the decomposition from the final path.

References

I. Davis, *A Modular Neural Network Approach to Autonomous Navigation*, Ph.D. thesis, Carnegie Mellon University, 1996.

A. Kelly, *An Intelligent Predictive Control Approach to the High-Speed Cross-Country Autonomous Navigation Problem*, PhD Thesis, Carnegie Mellon University, 1995.

J. C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Boston, MA 1991.

F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

R. Sedgewick, *Algorithms in C++,* Addison-Wesley, 1992.

W.B. Stout, "Smart Moves: Intelligent Path-Finding", *Game Developer Magazine*, October 1996.

S. Tanimoto, *The Elements of Artificial Intelligence*, Computer Science Press, Rockville, Maryland, 1987.

Acknowledgments. The author would like to thank Activision and Paramount Pictures for the opportunity to work on this project and for permission to publish results. In particular, Scott Lahman, VP of the Activision LA Studio has fostered an environment which encourages advances in technology.