Hybrid Vector Field Pathfinding

Johannes Moersch

Howard Hamilton

University of Regina

1. Introduction

Pathfinding is an important aspect of almost all of today's interactive entertainment. Whether it is cars finding the route around a track, enemy (or allied) soldiers moving through a building or across a field, or hordes of minions running across large tracts of landscape, pathfinding algorithms are what allow these interactions to occur. Many methods of pathfinding (both in terms of search algorithms and data representations) are in use today, but most of these methods are built for pathfinding through static environments, and consequently, are poor at dealing with moving objects, which are called ***dynamic objects*** in this context. Often hacks are employed to help with short term object avoidance, but these methods do not solve the problem of long term planning for dynamic objects. In short, individual pathfinding is a largely solved problem, but crowd pathfinding has a long ways to go.

2. Background

2.1 Current Techniques

Most current pathfinding implementations uses the ***A* algorithm*** (or a variant of A* optimized for some purpose), but despite this commonality, a wide variety of different implementation techniques are used. A* itself is a recursive search. It starts with knowledge of only a start point and an endpoint, and from there finds neighbours and chains them to reach

the specified goal. The neighbour to go through next is selected from the neighbours of the known point with the lowest combination of distance from start point and estimated distance to endpoint. The performance and accuracy of this algorithm can change dramatically with different world space data representations, but regardless of this fact, there is only so much this algorithm can achieve [Cui 2011]. It is still limited to searching through a space that lacks time. Theoretically, time could be added to the search space as an additional dimension, but this would likely be highly inefficient. Instead, dealing with dynamic objects is left to systems that are (in a sense) hacks that get tacked on. These hacks modify the already found path to deal with local (unaccounted for) obstacles such as other moving entities. These methods help, but they are unable to create realistic crowd motion.

Overall, there is a significant amount of research on creating realistic crowd motion, but many of the existing methods (like A*) are agent based methods, which means that each decisions about paths are made individually for each entity. While this approach is fundamentally sound, achieving realistic results with it appears complex and computationally expensive. Real people are continually gauging the speeds of dozens of people, predicting their paths of motion, and in a sense, planning their path in 4D (through time). Much of the existing research attempts to simplify what people are continually doing down to a set of algorithms and precomputed data that is manageable for real time use, but they have had limited success [Treuille 2006].

2.2 Examples of Problems

The subject of crowd pathfinding is particularly important for real time strategy (RTS) games because these games deal with large groups of units. If traditional pathfinding techniques are used, an attempt to move two groups of units through one another results in a mess.

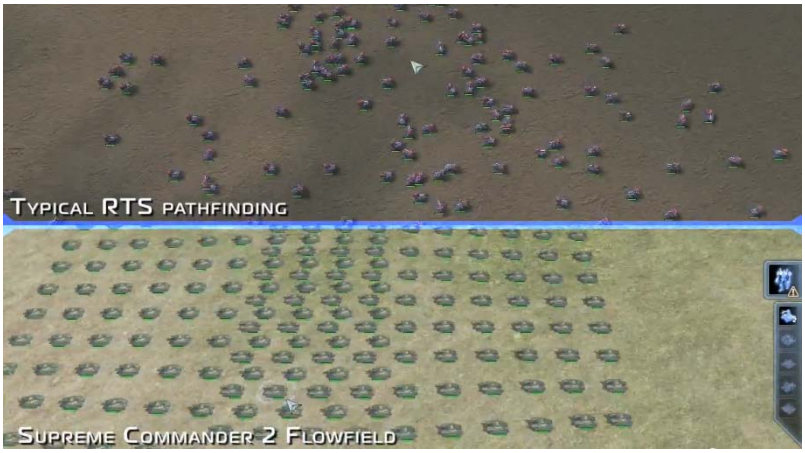

Figure 1. Screenshot from a promotional video for the RTS game Supreme Commander 2 [Gas Powered Games 2010]

As can be seen in Figure 1, Supreme Commander 2's flow field technology (which is based on "Continuum Crowds" [Treuille 2006]) is a workable solution to this problem.

Additionally, there is the problem of congestion. When there are multiple routes to a destination, sometimes not all units should choose the shortest path because unit congestion could actually make the shortest route take more time to traverse than a longer alternative. The approach presented in this paper attempts to address these two problems in a way that is applicable to real world usage.

3. Approach

To create a flexible real-time solution to group pathfinding, the problem can be divided into two distinct parts: long-range pathfinding for global route calculation and medium-range pathfinding for dynamic object avoidance. The solution to the long-range subproblem is the tried and true A* algorithm, and the solution to medium-range subproblem is a novel technique based on "Continuum Crowds" [Treuille 2006].

3.1 Long Range Pathfinding

There are two reason why the A* algorithm is used for long range pathfinding. First, A* is a proven algorithm whose primary flaw is an inability to deal with dynamic objects. Since the medium-range algorithm will deal with dynamic objects, this flaw is not relevant. Second, with an appropriate data representation, A* can be tuned to achieve a very high level of performance and accuracy.

3.1.1 Data Representation using a Navmesh

The most basic and naive data representation of world space is a simple uniform grid. However, this approach is a poor choice for large open areas because of its inefficient use of memory space. A simple improvement is to use a quad tree, or another representation that supports representing large uniform regions as a single data point. However, at least in first person shooters, grid based approaches are rarely used. Early first person shooters used a representation based on waypoints. Waypoints are interconnected points in space and the lines that connect waypoints are considered traversable. This method, while efficient, is severely lacking because it provides no information about whether the space directly next to the path is

traversable, so (unless also using local pathfinding) entities are essentially locked on rails. Today, waypoints have been almost entirely abandoned in favour of other techniques.

A common current technique is to use a ***navigation mesh*** (***navmesh***), which is a collection of convex polygons that represent the traversable space. In this representation, 2D (and theoretically 3D) regions can be represented, and consequently freer (non-rail-locked) paths can be found. Navmesh's support for arbitrarily shaped and sized regions makes it especially suited for representing large open areas [Tozour 2008]. For this reason, the navmesh was selected as the data representation for this project.

3.1.2 Construction of a Navmesh

The most straight forward approach to creating a usable navmesh is to hand-create it, but the time cost is high. Instead, an algorithm was devised to dynamically remove convex regions from an already existing navmesh. The algorithm begins with an initial navmesh consisting of a single polygon representing the complete 2D space to be represented. Then the algorithm adds the obstacles, each modeled by a polygon, one by one. When an obstacle is added, it is either completely inside the remaining open area or overlaps part of it. We treat the problem as a series of problems of removing one convex polygon from another.

Each convex polygon can be regarded as a clockwise loop of vertices around an interior point, so from one point of view, the algorithm needs to break one convex loop into several. The simplest way to assure that all the resulting polygons from subdivision are also convex is to project a line out from each removal point that resides inside the current polygon. The projection method is illustrated in the following diagram. The initial navmesh is shown in blue

and the obstacle to be added is shown in white. The obstacle is subtracted from the existing navmesh. Two vertices of the obstacle are inside the navmesh, so two lines are projected outward across the navmesh. The remaining part of the navmesh is split along these lines into a total of three polygons.

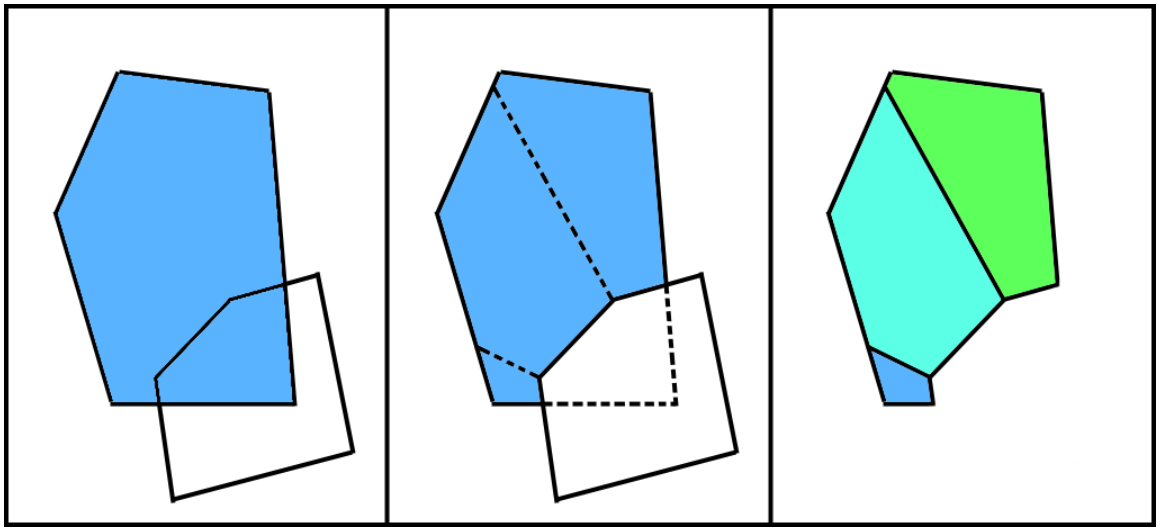

Figure 2. Navmesh Subdivision Example

The concept is simple, but implementing it algorithmically is a little more difficult. The approach used is to loop through all the sides of the shape to be removed, and for each side perform the test shown in Figure 3.

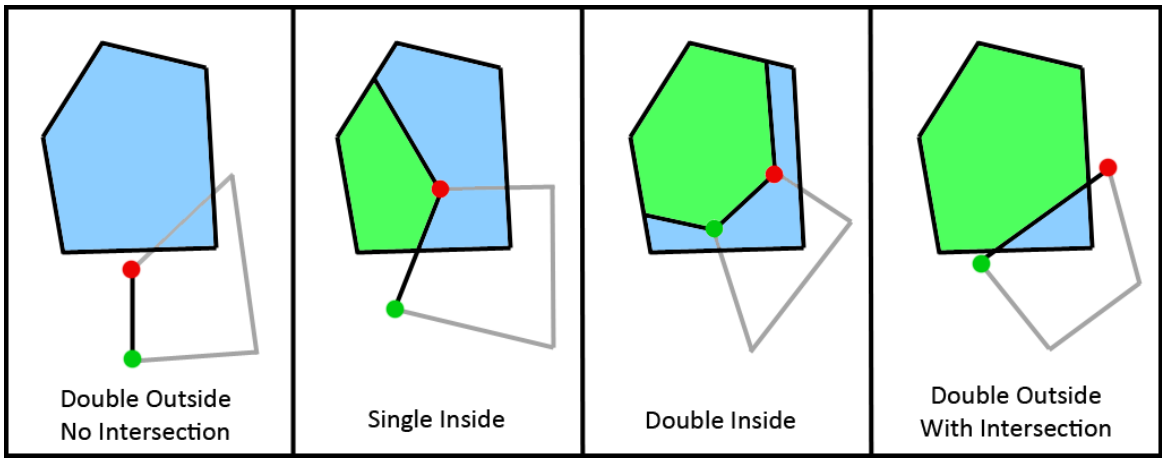

Figure 3. Navmesh Subdivision Cases

In the case where both end points for the given side are outside of the navmesh and the line between them does not intersect the navmesh, no polygons are generated. In all other cases, one polygon is generated. The shapes of the generated polygons in the other cases are suggested by Figure 3. In each case, the polygon that is generated is marked in green in Figure 3. If a polygon is generated, then discard the original, and replace it with the new ones. If none is generated, then perform a check to see whether the original polygon is entirely contained by the removal polygon. If so, simply discard the original.

Since A* requires neighbour information, every time you subdivide, you must also recalculate neighbour information. The first step is to set neighbour information for adjacent polygons among the newly generated polygons. This can easily be done by checking for consecutive cases of polygon generation. If two are generated in a row, they are neighbours. It is important to also check if the first and last side both generated a polygon, because if they did, they are also neighbours. Then, for each new polygon, iterate through all of the neighbours

of the original polygon and look for shared sides. If you find a shared side, that polygon is a neighbour.

3.1.3 Traversal

The A* algorithm depends on the ability to calculate the cost to travel from one node to any of its neighbours. Calculating this cost is trivial for a uniform grid, but this calculation is significantly more involved when dealing with navmesh. Consider the situation where you have a piece of mesh that is shaped like a trapezoid with a very short top and a very long bottom. Traveling from left to right near the top with is a short trip, whereas traveling from left to right near the bottom is a long trip. With navmesh, there can potentially be huge variation in the cost to traverse a polygon, and because of the nature of the search technique, there is no practical way to predict which distance is most accurate. To mitigate this inaccuracy, one solution is to subdivide the world into a coarse grid before removing anything from the mesh. This sets a limit on the inaccuracy of navmesh while mostly retaining its good use of space. However, this still does not answer the question of how to calculate the cost of traversal.

There are numerous solutions to cost calculation, but an easy and effective one is to calculate the cost to move between the midpoints of the two connecting sides.

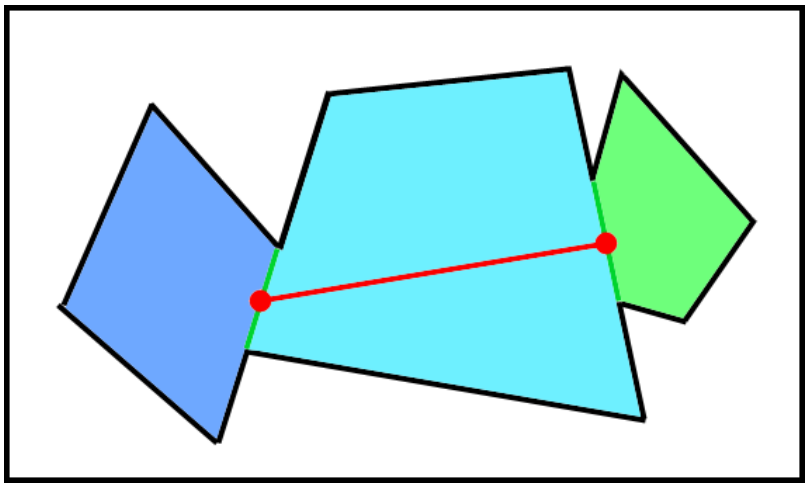

Figure 4. Navmesh Traversal Cost

As can be seen in Figure 4, the midpoint along a connecting side is found from only the overlapping region between the two polygons, not the entire side. The rest of the long-range pathfinding implementation is standard binary heap based A*.

3.2 Medium Range Pathfinding

Since the long range pathfinding does not deal with dynamic obstacles, this task falls to the medium range pathfinding. The algorithm presented in “Continuum Crowds” is tailored to just this task, and for this reason, I selected it.

3.2.1 Concept

The approach taken by Treuille attempts to tackle these problems by treating pathfinding as flow field problem. Instead of pathfinding for every individual unit separately, the idea is to group units based on destination and then generate a single flow field for each group. This flow field defines the optimal direction of movement for a unit at any location in the given area. The entities can then simply sample this flow field to find their direction of motion.

The goal of the generation algorithm is to create a map of the minimum cost from any given location to a specified destination. This map can be seen as a uniform 2D grid containing values acquired from a ***potential function***, which gives the cost to the goal from a point in R^2. Since cost increases as you move further from the end goal, following the negative gradient of this function will move you towards the goal.

To calculate the potential function values, some other maps must first be generated. To start, a density map and a velocity map must be calculated. The density map is generated more or less by simply "splatting" units onto the map as circles of density that fall off away from their centers. As this map is generated, unit velocities are also calculated and placed in a different map. Then, using these maps as a data source, a speed and a cost map are generated. Both of these maps are ***anisotropic***, meaning their values depend on direction. One can picture an anisotropic map as either a grid of cells with four values in each cell (one for going toward each of +x, -x, +y, and –y) or one can picture four separate grids, one for each direction. The speed map holds unit flow speed and the cost map holds cell traversal cost. Then, using the fast marching method, the potential field can be found, and finally the vector field (which contains the negative gradient of the potential field) can be calculated [Treuille 2006].

3.2.2 The Equations

3.2.2.1 Density and Velocity Maps

The ***density map*** gives a real value for each grid cell representing the density of units in that cell. Calculating the density map is a relatively simple process. Each unit contributes to the density of all grid cells within a circle around its center. The contribution to each grid cell within

this region falls off linearly from the unit's center. This value should be clamped to be non-negative. The density contribution to cell i from unit j is described by the following equation.

$$density_{ij} = 1 - \frac{|position_i - center_j|}{radius_j}$$

where $position_i$ is the position of cell i and $center_j$ and $radius_j$ define a circle centered on unit j.

Velocity is similarly simple to calculate. The ***velocity map*** holds the velocity at each cell i. It is calculated by taking a weighted average, depending on the density, of the velocities of nearby units. The follow equation shows how the velocity is calculated.

$$velocity_i = \frac{\sum_j (velocity_j \times density_{ij})}{\sum_j density_{ij}}$$

where $velocity_j$ is the velocity of unit j. The summations can be performed at the same time as the density map calculation; however, the final divide by the total density must wait until after the influence of all units has been calculated.

3.2.2.2 Speed and Cost Maps

The speed and cost maps are both anisotropic, so for each grid cell four values have to be calculated: one value for each of +x, -x, +y, and -y. For the speed map, the (say) +x value represents the flow speed when moving in the same direction as the positive x axis. This value is a scaling factor in the range (0, 1] that will be applied to a velocity component in the corresponding direction. For the cost map, the +x value represents the cost of travelling from the current grid cell to the neighbour in that direction.

Before calculating the speed map, upper and lower density thresholds have to be chosen. Suppose that we are considering one cell, called the ***current cell***, of the speed map for the +x direction. We start by picking the ***destination cell***, which is a cell in the +x direction. Let *offset* denote the destination cell. The destination cell may be the immediate neighbour or one farther away. We look at the density and velocity for that cell. If the density of the destination cell is below the lower threshold (ρ_{min}), the flow speed for the current cell is set to the maximum (1), which means that a unit can go at full speed through this cell because there are no obstacles nearby. If the density for the destination cell is above the upper threshold (ρ_{max}), the flow speed for the current cell is the dot product of the velocity in the destination cell $velocity_{offset}$ and the unit direction vector n_{offset} for this map (e.g., (1, 0, 0) for the +x direction), clamped to the range [0, 1]. If the density is between the thresholds, the flow speed is interpolated linearly between these values. The equation for flow speed is as follows.

$$speed = 1.0 + clamp\left(\frac{density_{offset} - \rho_{min}}{\rho_{max} - \rho_{min}}\right)\left(clamp\left(velocity_{offset} \cdot n_{offset}\right) - 1.0\right)$$

where n_{offset} is a unit length vector that points in the direction being calculating, $density_{offset}$ is the density of the neighbouring cell in that same direction, $density_{offset}$ is the velocity of the neighbouring cell, and *clamp* is a function that restricts the range to 0 to 1 inclusive.

The cost calculation is a simple step from here. The cost is calculated as follows.

$$cost = \frac{(speed * \alpha + \beta)}{speed} = \alpha + \frac{\beta}{speed}$$

In this equation, α and β are weight factors. For the experiments, these were set to 0.1 and 0.9, respectively. Of course, varying these factors affects the cost. Let us consider an example to

see the effect of using the equation. If the density is low, then speed is 1.0, and thus, the cost is 0.1 + 0.9/1.0 = 1.0 (minimum cost). Similarly, if the density is high and most units in the destination square are going the same direction (say +x in our case), then speed is still 1.0 and the cost is 1.0. On the other hand, if density is high and most units in the destination square are going in a different direction, then speed is low, say 0.2, and the cost is 0.1 + 0.9 / 0.2 = 0.1 + 4.5 = 4.6. This high cost will discourage A* from creating a path through the destination cell.

3.2.2.3 Potential Field

The ***potential field*** gives the cost to the goal from every cell in the grid. There are two primary algorithmic components to calculating the potential field. The first is how to calculate the lowest cost to move into a cell, and the second is how to traverse the data when doing the calculations.

To find the minimum cost of entering a given cell, we first find which cell(s) we are entering from. The candidate cells are the cells +x, -x, +y, and –y. If both cells on a given axis have an undefined potential value (because they have not yet been calculated or are unreachable) then choose neither and only select a cell from the other axis. In either case, the cell to choose is the one with a lower value resulting from the following equation (where *cost* is the cost to move from the current cell to the cell in the given direction, as obtained from the cost map).

$$value = potential_{offset} + cost$$

Once the cells of entry have been selected, the next step is to solve for *potential* in the following equation. This value is the new potential value for that cell.

$$\left(\frac{potential - potential_{offsetX}}{cost_X}\right)^2 + \left(\frac{potential - potential_{offsetY}}{cost_Y}\right)^2 = 1$$

If only one direction of entry exists (e.g., along say the X dimension), just drop the other term from the equation. To solve the equation, expand it out and put it into quadratic form. Then use the quadratic equation to solve it. If there are two solutions, choose the larger of the two.

Next, to apply this equation, one should use the ***fast marching method*** [Sethian 1999; Baerentzen 2001]. The fast marching method is based on Dijkstra's algorithm, which is similar to A*. The three major differences between it and the A* algorithm is that (1) with the fast marching method one begins with the goal instead of the starting location, (2) one only terminates when the entire neighbour list is empty rather than when the goal is found, and (3) one must keep track of the total cost to reach each cell and output this into the potential field [Shopf 2008]. As with A*, a binary heap should be used as your priority queue.

3.2.2.4 Vector Field

To calculate the ***vector field*** you need to find the gradient of the potential field and multiply it by the appropriate scaling constants from the speed map. To find the negative gradient, evaluate the following equation.

$$vector = \frac{vec2(potential_{-X} - potential_{+X},\ potential_{-Y} - potential_{+Y})}{|vec2(potential_{-X} - potential_{+X},\ potential_{-Y} - potential_{+Y})|} * vec2(speed_X, speed_Y)$$

In this equation, $speed_X$ and $speed_Y$ are the speed values at the current cell in the direction which the gradient faces on each axis and * represents component-wise

multiplication. The vectors that are calculated here represent the velocity a unit within a given cell should follow.

3.3 Joining the Two Techniques

The two techniques are combined in a simple layered way. The long-range pathfinding is used to find an intermediate destination for the medium-range pathfinding, and then the medium-range pathfinding is used to guide the actual units. The method begins by placing units that share a common destination in a group. For each group, the method finds an approximate starting point (by finding the unit closest to the average position) and then finds a path from there to the goal using the A* algorithm. Then it selects a region for which to calculate a vector field and calculates it. This region is selected by finding the bounds of the group of units and adding some arbitrary amount to each side. To find the ***destination point*** for the vector field calculation, the algorithm steps along the constructed A* path until it reaches the end point of the path or leaves the region. The end point or exit point from the region is the destination point for the vector field.

Figure 5 shows an A* path for the group of white units to the upper right of the picture and the potential field generated to influence the movement of these units along the path up to the destination point, which is the leftmost point along the white line inside the potential field.

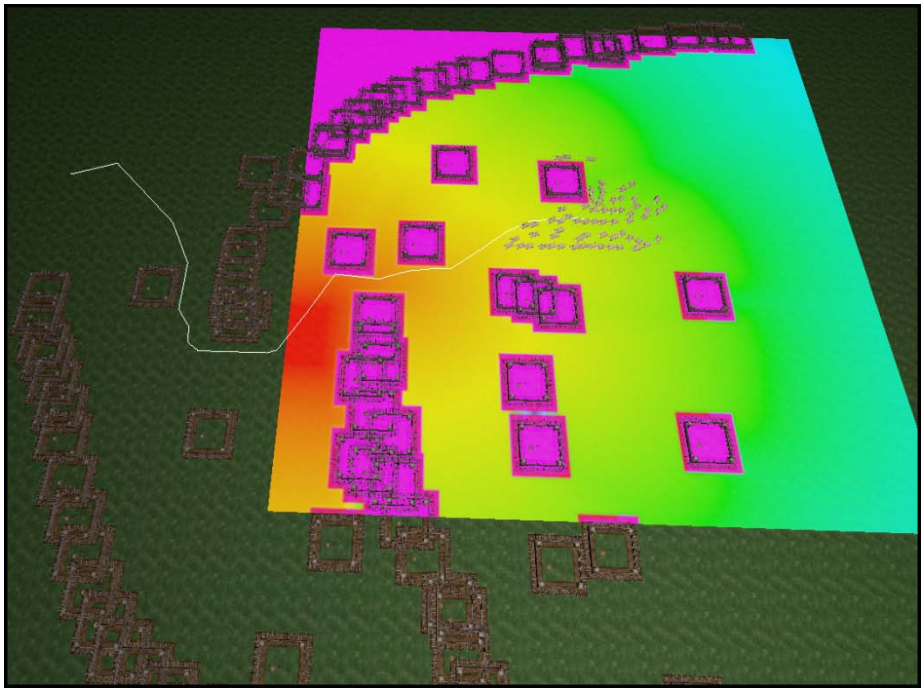

Figure5. A Path and Potential Field Combination*

Once the vector field has been calculated, the ***target velocity*** for each unit is obtained by taking an interpolated sample for it. This sample is obtained at the unit's ***expected location***, which is the location where it is expected to be in 0.75 seconds, assuming it does not change its course. Using the expected location rather than the actual location allows the units to react more quickly and significantly improves the results. The interpolation is performed using ***bilinear interpolation*** from the four velocity vectors at the corners of the cell in the vector field that contains the expected location.

A tweak that reduced the number of visual artifacts was to enforce grid alignment. In particular, the grid cell size was held constant and every region was aligned along some grid cell boundaries. This enforced grid alignment prevented flicker, which could otherwise cause problems when pathfinding through narrow gaps. Also, because of the high cost of the

potential field construction, pathfinding should be performed asynchronously with respect to graphical display. Otherwise, running the algorithm will cause noticeable stutters in the video output.

4. Results

4.1 Lane Formation Test

Lane formation is an important part of group dynamic object avoidance and an established advantage of the “Continuum Crowds” algorithm. The results observed in our experiments for lane formation were mixed. While lanes do form, many unit collisions occur. The results can be seen in Figure 6.

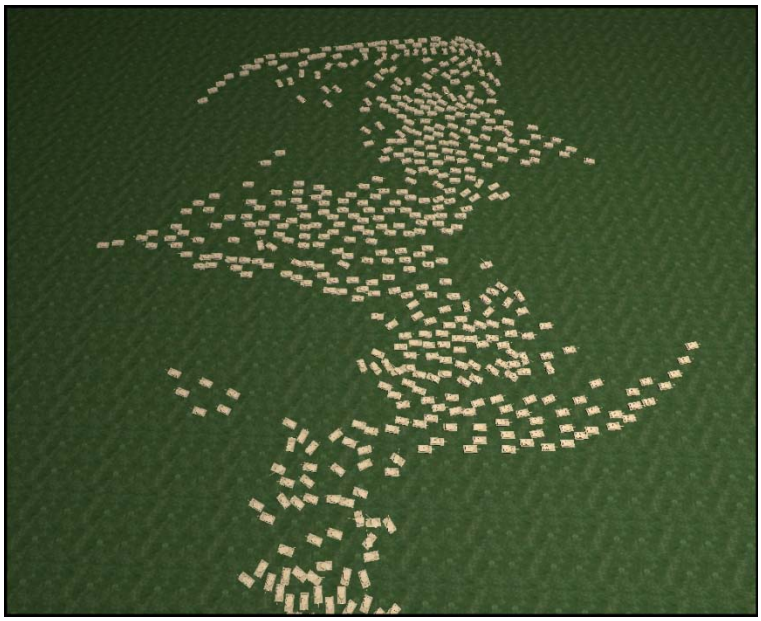

Figure 6. Lane Formation Test

As can be seen by comparing Figure 6 to Figure 1, better results can be achieved. However, a trade-off is observed when trying to improve this performance. Three factors that are a detriment to the results are that the units have a wide turning radius, the units are very difficult to push out of the way, and the units have somewhat low acceleration rates. All three of these things make the units less reactive and more likely to get held up by collisions than the units in Figure 1. This can be considered a trade-off because these same features make the units move more realistically. That being said, the overall effect is still far superior to that of traditional pathfinding techniques.

4.2 Congestion Avoidance Test

Congestion avoidance is arguably the most desirable attribute of a medium-range pathfinding algorithm, and the congestion results for the approach described here are promising. All the units in Figure 7 started in a group on the right, and were told to form a new group on the left.

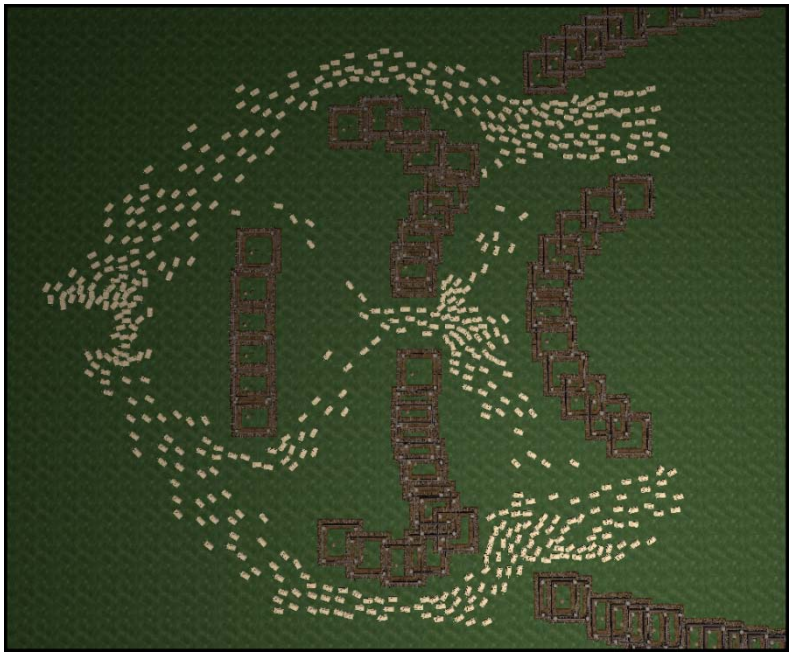

Figure 7. Congestion Avoidance Test

As can be seen, the units are choosing multiple paths to the destination. When viewed in motion, the units can be seen reacting to choke points and choosing new paths as the choke points become blocked with other units.

4.3 Maze Test

The maze test tests the long-range pathfinding capabilities. As can be seen in Figure 8, the hybrid method works well.

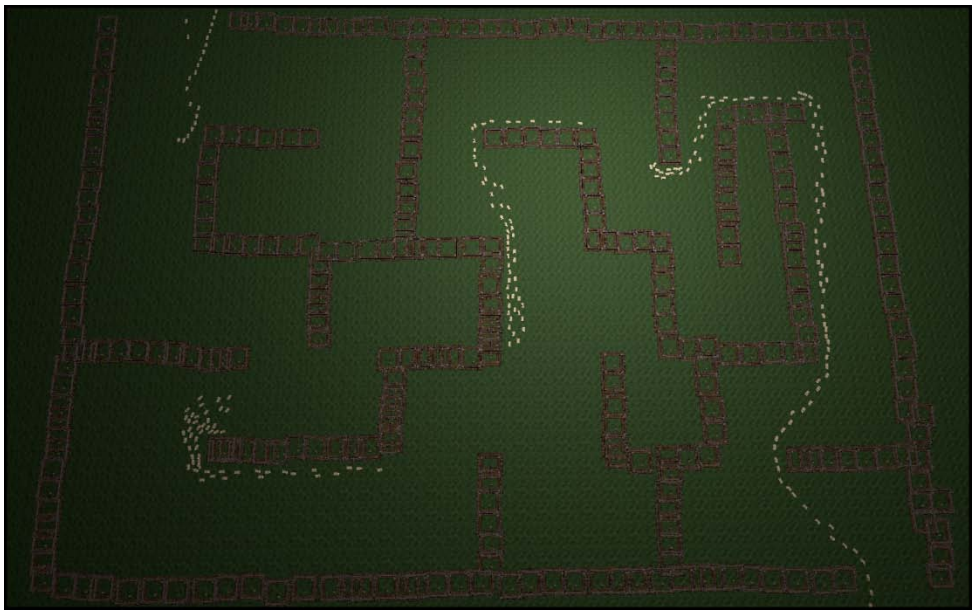

Figure 8. Maze Test

Overall, the hybrid technique works very well. However, when travelling longer distances the groups can become quite spread out which can cause the medium-range pathfinding region to become very large. In future work, this may be remedied by automatically subdividing each group into smaller groups when they spread out too much.

4.4 Performance

The high cost of the medium-range pathfinding algorithm makes performance one of the primary concerns when considering this technique. Since most of the cost is in the execution of the fast marching method, the number of units has little bearing on the performance. The primary performance factor is the grid resolution, i.e., the number of cells in the grid. The execution times for a full long-range/medium-range pathfinding query on a non-complex scene at different resolutions can be seen in Table 1. The execution time is the time to perform the long-range pathfinding by building the navmesh and executing the A* algorithm on it plus the time to perform the medium-range pathfinding by constructing the density and velocity fields, the speed and cost maps, and the potential and vector fields. We used square grids with s^2 cells for a side length of s; for example, a size such as 64 gives 64 x 64 = 4096 cells. The ratio is determined as the number of cells calculated per millisecond; for example 4096 cells calculated in 4 ms gives 1024 cells/ms.

Side Length of Grid	64	128	192	256
Cell count	4096	16 384	36 864	65 536
Execution Time (ms)	4	18	37	65
Ratio (cells/ms)	1024	910	996	1008

Table 1. Execution Times

This table shows an approximately linear cost relative to the grid cell count. These costs are too high to match the rate of real-time graphics, but with the queries running asynchronously and at a lower frame rate than the graphics, the cost is manageable. Whether the cost is justified would depend on the application.

5. Limitations and Improvements

There are some improvements that could be made to the A* data representation, such as mesh optimization and more accurate traversal cost calculation, but these concerns are not the focus of this paper. The focus lies more on the medium-range pathfinding technique, and it could certainly be improved. In its current implementation, the algorithm has no support for varied or sloped terrain and it does not support discomfort fields, both of which have previously been addressed [Treuille 2006]. However, adding both of these features would be straightforward. Additionally, there is room for tweaking and optimization.

Currently, the potential field will grow to cover a region as large of the entire group of units. It does not consider whether the units are spread out across the whole map. An automatic group subdivision technique would be highly beneficial. One might also be able to automatically join groups that have endpoints that are sufficiently near to each other.

6. Conclusion

The vector field pathfinding concept [Treuille 2006] shows a lot of promise. While computational cost and scale are severe limitations of the algorithm, both of these can be overcome by combining the technique with other pathfinding techniques such as A*. A well-developed hybrid system (such as the one used by Gas Powered Games in *Supreme Commander 2*) could be effective in guiding large numbers of units around complex environments. Hybrid techniques with vector fields for pathfinding have a lot of potential and are definitely worth researching further.

7. References

Baerentzen, J. A. “On the Implementation of Fast Marchine Methods for 3D Lattices,” Technical Report IMM-REP-2001-13, Technical University of Denmark, 2001.

Cui, Xiao. “A*-based Pathfinding in Modern Computer Games” IJCSNS January 2011. <http://paper.ijcsns.org/07_book/201101/20110119.pdf>

Gas Powered Games. “Supreme Commander 2 ‘Flowfield Pathfinding’ Trailer” Youtube 2010. <http://www.youtube.com/watch?v=bovlsENv1g4>

Sethian, J.A. “Fast marching methods,” *SIAM Review*, 41(2):199-235, 1999.

Shopf, Jeremy. “Crowd Simulation in Froblins” SIGGRAPH 2008. <http://s08.idav.ucdavis.edu/shopf-crowd-simulation-in-froblins.pdf>

Tozour, Paul. “Fixing Pathfinding Once and For All” ai-blog.net 2008. <http://www.ai-blog.net/archives/2008_07.html>

Treuille, Adrien. “Continuum Crowds” SIGGRAPH 2006. <http://grail.cs.washington.edu/projects/crowd-flows/continuum-crowds.pdf>