Formation-Based Pathfinding With Real-World Vehicles

Jim Van Verth
Victor Brueggemann
Jon Owen
Peter McMurry

Red Storm Entertainment
2000 Aerial Center, Suite 110
Morrisville, NC 27560
{jimvv,victorb,jono,peterm}@redstorm.com
(919) 460-1776

Abstract

A number of papers and articles have been written about formation-based pathfinding. Many of them, however, make the assumption that the units involved can move in any direction and can turn on a dime. This paper presents our solution in Force 21 to the problem of controlling real-world vehicles in formation, what we learned from it, and what we will do differently the next time.

An alternative algorithm is presented, which makes use of a pre-computed visibility graph and Dijkstra's shortest path for the broad movement strokes, per-platoon morphing formations for moving along that path, individual vehicle AI for moving to formation position and Reynolds' steering behaviors for avoiding dynamic objects in the world. The tricky cases of maintaining temporary bridges and roads (faster, shortcut paths) in this environment are also covered.

1 Introduction

Pathfinding has been a problem in a broad range of disciplines including robotics, artificial intelligence and artificial life. Some of the earliest algorithms involve searching a space, whether it is defined by a graph or path-planning for a robot. Nearly all games that involve maps and autonomous units require some sort of pathfinding. Tactical and strategic planning can be finessed through peeking at the player's units or other means of cheating, if necessary. It's also the aspect the player will see the most often in a strategy game, since he or she will be manipulating units indirectly and expecting their AI to behave and move in an intelligent and snappy manner. Therefore pathfinding is the most exposed and the most difficult part of the AI system.

The algorithms presented in this paper are the result of the work of many different people over three separate projects. The first pass at the basic visibility graph algorithm was done for Planet Texas by Peter McMurry. It was later revised and put into a form suitable for Force 21's

predecessor, Dominant Species, by Jon Owen. Force 21 was built on Dominant Species' terrain, graphics and to some extent simulation engine and so also inherited the pathfinding methodology. Various extensions to the algorithm were needed for Force 21's particular needs – that work was done by Jim Van Verth and Victor Brueggemann.

Related work

Other solutions to the problem exist, and were examined prior to starting our approach. The most commonly used path-determination algorithm is A*search. [Stout] provides a good overview of the general A* search algorithm, and [Vinke] discusses a real-time implementation. It works on the principle of searching for a local minimum based on a heuristic function. As such, it is only as good as the heuristic used, and as it is looking for minima if implemented poorly it can get bogged down quickly. It is also only good for static environments and very poor for searching large spaces. There is also a variant of A* called D* which purports to be good for searching dynamic environments, but it gets very expensive with increasing number of elements.

A first pass at A* was attempted in Dominant Species but it was decided that it was computationally too expensive and not necessary because terrain slope was not a factor in the movement of the creatures. Force 21 inherited the Dominant Species search algorithms and attitude. Later changes in game design – namely terrain slope affecting vehicle speed – would have benefited from some sort of A* search, but there was no time in the schedule.

The next stage of the problem, moving the units along the path, is presented by Dave Pottinger in [Pottinger]. Pottinger's approach encapsulates both the problem of arranging units in a tactical group, collision detection, and the actual movement of them through the world. However, his algorithms are best suited for non-realistic games with large groups of units (100+). Our system has a physically-based engine for moving objects through the world and so his algorithms are not entirely appropriate. However, some of the techniques presented were incorporated into our system, in particular the notions of high-level and low-level pathfinding, and some of the aspects of formation and collision management.

2 Framework

Before presenting our algorithm, some basic concepts about our simulation engine and design are necessary for proper understanding.

Force 21 is real-time strategy game in a 3D environment, with collections of autonomous moving objects or *vehicles*. There are three main classes of vehicles. The first are *tracked* vehicles, which are primarily battle tanks and the like. These vehicles move slowly, but can move one set of treads forward and one backward, and so turn in place. The second are *rotary wings*, which are helicopters. They move quickly through the air, and like the tracked vehicles can turn in place. The last are the *wheeled* vehicles, which are fast-moving ground vehicles, but have a limited turning radius, and cannot turn at all when stopped.

Each vehicle has physical characteristics, most particularly vectors representing *position* and *velocity*. The simulation engine is a physically-based system which controls how the vehicle

and other physical objects move through the world based on their physical characteristics. Part of this is a real-time collision system, which detects collisions and computes appropriate response velocities. Objects are moved by either directly setting their position, or setting an *acceleration* vector, which over time will change object velocities and thus their positions.

Players do not control individual vehicles. Instead they issue orders to *platoons*, which are a collection of up to five vehicles. Platoons also have their own position, which is generally the location of the head vehicle. The platoon AI or *tactician* interprets the orders and issues appropriate commands to each vehicle's AI or *crew* through a message-passing system. If necessary, the crew then sends driving commands to each vehicle.

Vehicles in a platoon are arranged in a *formation*, each of which is designed for a particular situation. The four formations are shown in Figure 1. As vehicles move across the landscape they are expected to maintain formation within their platoon as well as avoid obstacles and each other.

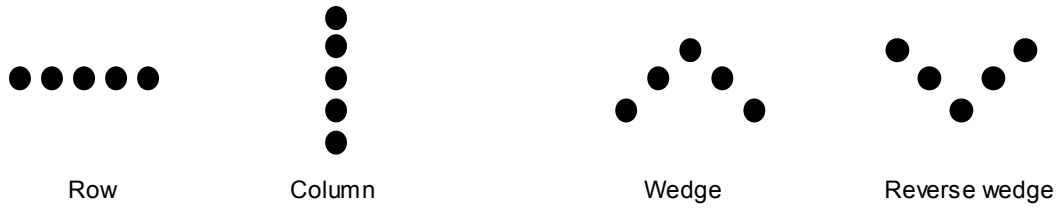

Figure 1: Four formation types

A general overview of the pathfinding algorithm is as follows. An order to move is issued to the platoon. The tactician for that platoon computes the shortest path across the map to avoid known static obstacles and then begins moving the platoon along it. While platoon moves its tactician issues commands to each crew telling it where to move to in order to maintain formation. Each crew in turn checks for other vehicles and computes a steering direction for the vehicle towards the desired formation position. The vehicle then computes driving accelerations appropriate to the vehicle type that best move the vehicle in the desired steering direction.

3 Platoon-level pathfinding

As mentioned above, the first stage in the pathfinding algorithm is computing a safe path for the platoon. The tactician considers only static objects, i.e. collections of trees or *forests*, buildings, and impassible terrain. The one exception to this is a bridge. Bridges can be added (by a bridge-layer) or removed (by destruction or a bridge-layer), but are still considered as part of the platoon-level pathfinding algorithm. This is discussed below.

Visibility graphs

The primary part of the algorithm is a minimum cost graph-search across a data structure called a visibility graph. Visibility graphs are computational geometry constructs originally

designed and primarily used for robotic motion planning. Good coverage on constructing a visibility graph can be found in [O'Rourke].

As an example of computing a visibility graph, take a top down view of a room, with polygons representing the obstacles in the room. The visibility graph for the room uses the vertices of the polygons for the vertices in the graph, and weighted edges between the graph vertices where there is a clear path between the corresponding polygon vertices (i.e. a line between the two polygon vertices crosses no polygon edges). The weight of each graph edge equals the distance between the corresponding polygon vertices. To compute the shortest path from point *a* to point *b*, then, requires adding *a* and *b* into the graph with the corresponding visibility information, and then finding the shortest path in the graph.

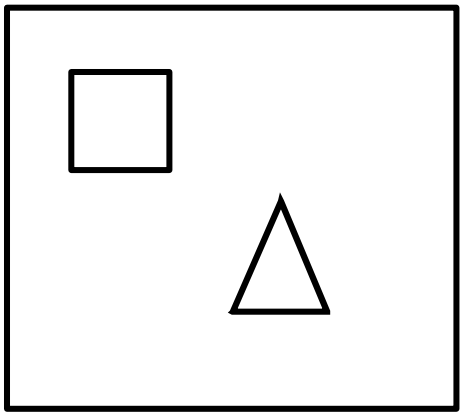

Figure 2.1: polygon map

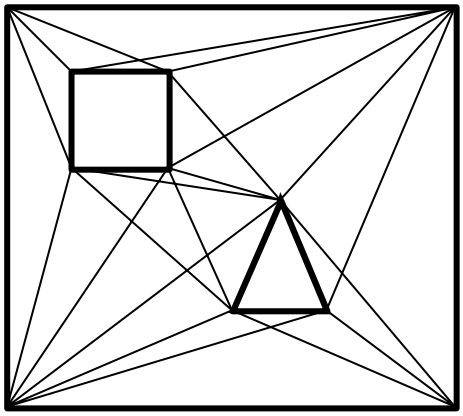

Figure 2.2: visibility graph

The outer boundary polygon is called the *container*. The inner polygons are called *obstacles*. There can be multiple container regions containing their own set of obstacles.

The shortest path is computed by using Dijkstra's shortest path algorithm for weighted graphs. [O'Rourke] is again a good source for a description of this algorithm. Other coverage can be found in [Even] and [Bondy].

Our heuristic

As mentioned above, the correct way to compute the path for two given points is to compute the visibility graph including those two points and then compute the shortest path from start to finish. Pre-computing the visibility graph can be done offline and loaded in at game time. The problem is adding the two search points. For a simple graph the cost of determining visibility information between a new vertex and all other vertices in the world is relatively low, but the cost can grow prohibitively as the number of nodes and the complexity of the scene grows.

To avoid this cost, a heuristic is used, which works in most cases. First of all, visibility is computed between the two search points. If there are no obstacles in the way, there is no need to search the visibility graph. If there is at least one obstacle, the intersection point with the first obstacle along the segment is computed and added to the graph. Links are added to the neighboring vertices on that polygon edge. Similarly, the last intersection point along the

segment is added to the graph. The search is then computed between the two intersection points (see Figure 3.1).

The resulting path is improved by checking to see if the edge between the start point and the second point in the path is clear. If so, the first point is thrown out and the second point is used. The new second point is considered, and so on, until a failure is detected. A similar check is done between the end point and the second-to-last point in the path. This removes unnecessary "kinks" in the path (see Figure 3.2). Appending the start and end points to the culled path creates the final path.

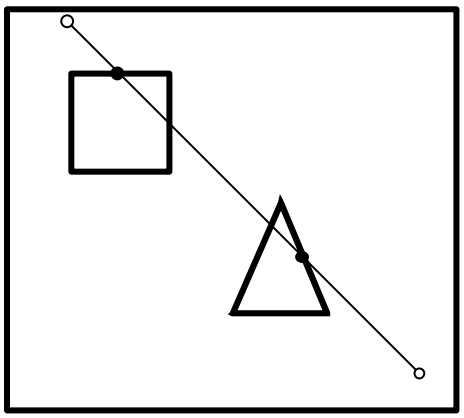

Figure 3.1: new start/end points

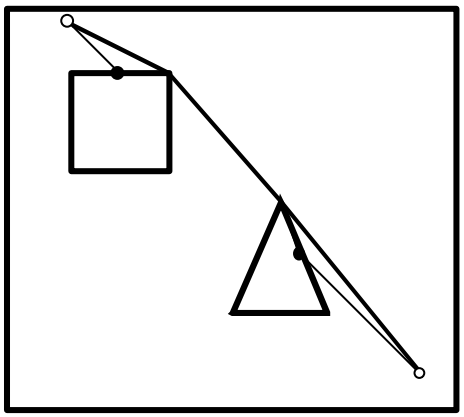

Figure 3.2: culling path

Buildings and forests

Incorporating this visibility graph into a scenario is very straightforward. Using our level editor, artists or designers mark off impassible terrain features with polylines. The level editor ensures that the polylines created cannot cross or enclose other polylines, which would create an invalid visibility graph. Upon closing the level, the editor creates the visibility graph based on those polylines and saves it as part of the terrain file. The terrain file is read into the game upon mission load and the corresponding visibility graph used for pathfinding. One additional change is that the path is slightly offset from each obstacle to allow vehicles and platoons to clear them (otherwise they would run into the corners of obstacles).

In our maps there are additional static obstacles that were not stored as part of the terrain information, primarily buildings and forests. We did not do this for two reasons: 1) because we do not load 3D information in our editor, we did not have the bounding boxes for the buildings and 2) this gave us a way to distinguish between terrain features and static objects. Instead, after the visibility graph file is loaded in, the game adds polylines for the forests and buildings. The kicker is that while the editor does not allow illegal polylines, a forest or a building boundary added later can potentially cross another polyline. An algorithm was devised that clipped a container if a non-container polyline crossed it, and created the union of two polylines if neither is a container polyline. If a building or forest polyline lies inside another non-container polyline, it is not considered at all. This way the artists can have chains of intersecting collections of trees and create one large forest.

Roads

The design called for vehicles to move faster on roads. If we had used an A* algorithm, this would have been relatively easy – just make the weights on the road vertices lighter than their surrounding vertices. However, since we were using a graph-based scheme, another solution was necessary.
Roads are a series of up to six vertices that are connected together to form a path. Joining roads together using road junctions can create longer paths, or Ts and crossroads. The level designer draws a road, places a road junction near the end of that road, and then starts another road nearby to join them together. This series of roads and road junctions forms the basis for both the graphics algorithm that draws the road, and the pathfinding algorithm.

A graph is created using just the roads and road junctions, but with a lower weight (1/2 the Euclidean distance) on the edges between vertices. This is stored in memory in addition to the original visibility graph. Once a path is generated using the visibility graph (and a map has roads), it is broken into smaller segments. Each vertex in the new path is compared to the road graph. If a path vertex is determined to be close to a road vertex, an edge is created connecting the two vertices together. After all the path vertices are checked, the shortest path is found on this new graph. If the path generated is shorter, it is used for the final path. To avoid overcomplicated paths, collinear vertices are removed before the path is returned.

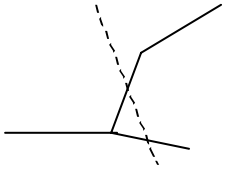

Figure 4.1: Road network and initial path

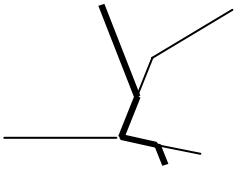

Figure 4.2: Road network and new path

Bridges

The design also called for bridges – bridges that can be added in the middle of the game, or removed (by destruction or bridgelayers). To add an additional complication, if a faction in the game doesn't know the bridge is gone, the pathfinding algorithm for that faction needs to act like it was still there, and only react accordingly when it was discovered that it was gone.

This is done in a similar fashion to the roads. A graph is created which includes all the bridges in the world. Each bridge takes up three vertices – two for each end, and one for the middle. Edges connect the middle vertex to the two end vertices. A path, using the main visibility graph, is computed between the ends of each bridge. If a path can be found from one end of a bridge to the end of another bridge, an edge is created between those two vertices with a weight equal to the length of the path. The path is stored for later use.

Again, like a map with roads, a path is computed using the visibility graph. Then, if there are bridges in the world, paths are computed to all the end vertices of the bridges in the world from both the start and end search points. If a path is found, an edge is created between the two vertices with weight equal to the length of the path, and the path is stored. Finally, a shortest path search from start to end is done on this modified graph, and if the newly computed path is shorter than the original, it is used instead.

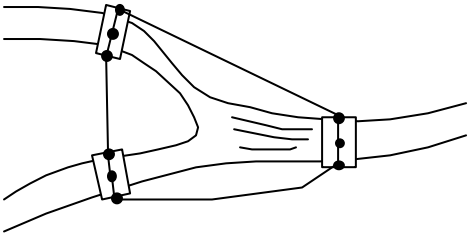

Figure 5.1: initial bridge graph

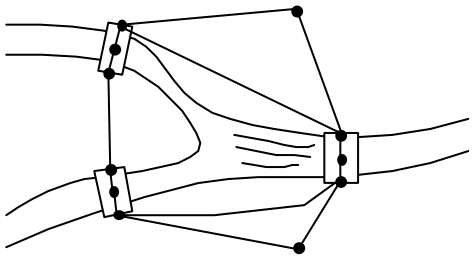

Figure 5.2: bridge graph with start and end nodes added

As one might guess, both the road and bridge pathfinding algorithms can be quite time-consuming, and so heuristics were derived to avoid calling them. For instance if a path was relatively straight, or relatively short, it was deemed unnecessary to continue further. On the other hand, if a visibility graph path doesn't exist, the start and end points may be in separate containers, and that makes the bridge search mandatory.

One other issue with bridges is that only one platoon at a time can be on them, since there is no room to pass. If a platoon is on a bridge, the bridge is marked with that platoon ID and other platoons must wait before crossing. Moving resting platoons to make way for moving ones was deemed to not be a major issue. Either the platoon on the bridge is a friendly unit, in which case it can be moved off by hand if necessary, or it is an enemy unit in which case it can be destroyed and the bridge is freed for crossing.

Swamps

Swamps are areas where the vehicles move slower over the terrain. This was built into the visibility graph by making some obstacles passable. Edges are constructed within the obstacle and given a higher weight than normal. In that way, if the paths through an obstacle and around it have approximately the same Euclidean length, then the best path will be around the obstacle. However, if the path around a swamp obstacle is very long, then the best path will be through it, even though the vehicle slows down briefly.

4 Formations

The above works fine for a platoon with a single vehicle. However, computing paths in this way for multiple vehicles in a platoon would be prohibitively expensive, they would not necessarily move in formation, and since they may move at different speeds would arrive at different times. A more structured means of moving a platoon of vehicles is necessary.

Formation movement

As mentioned above, each platoon is assigned a current formation. This is defined as a set of 2D vector offsets from the platoon's position, which are oriented to match the platoon's heading. The vehicle that lies nearest to the platoon's current position (i.e. an offset of zero) is known as the *lead vehicle*.

Instead of moving a single vehicle along the path, we move the platoon itself. As it moves, the heading of the platoon aligns itself with the current segment of the path. When it approaches a new segment, it begins to turn to align itself with that new segment. Note that the final heading of the platoon could be different from the alignment of the final segment, so the platoon makes one last heading change to match the desired heading (see Figure 6).

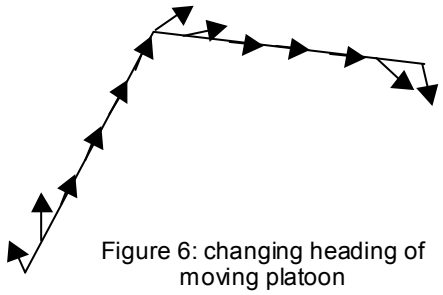

Figure 6: changing heading of moving platoon

We could have moved the lead vehicle along the path and followed its position, except that platoons in Force 21 are fixed – a vehicle does not get promoted up as other vehicles are removed or destroyed. The means that the lead vehicle position may be empty, so a separate position for the platoon is needed.

Once the platoon has moved to a new position and heading on the path, the tactician sends the vehicle's crew a message telling where it should be in order to maintain formation with rest of the platoon. Note that this is generally not the actual position of the vehicle, except perhaps when the platoon is at rest. Instead it represents a point for the crew to drive to, hence the term *drive-to* point. Below we discuss how the crew steers for that point, but suffice it to say for now that it tries its best to turn towards and arrive at its assigned goal.

Because vehicles have different speeds, and may need to slow down to avoid obstacles, the platoon cannot move forward at a constant rate. The initial speed of the platoon (*maxspeed*) is set to the speed of the slowest vehicle. The current position of each vehicle is then compared with its current drive-to point. The platoon will progressively slow down as the slowest vehicle drifts further and further behind and eventually reach a steady state. Eventually if the distance to the furthest vehicle surpasses a certain limit (2*maxspeed), the platoon stops and waits for the straggler to catch up.

When a formation is changed, even while moving, the new drive-to points are immediately computed. Because of this, there may be a slight delay in movement while the vehicles jockey into their new positions. In retrospect, it would have been better to gradually interpolate between the old positions and the new ones.

Column formation

Column formation is a special case. In column formation, the vehicles move in a line, with the second vehicle following the lead, the third following the second, etc. Originally this was done in similar manner to the other formations, with an offset computed directly from the platoon

position. The problem is that when the platoon turns a corner in the path, it looks very unnatural – the vehicles wheel out from their positions and then lock on to the new direction. Also, the last vehicle has to move the farthest distance and usually ends up out of position quickly, so that the platoon has to wait for it to catch up. Instead, the desired behavior is that all the vehicles stay on the path and follow it no delays, forming a snake-like pattern.

The process for determining the positions for column formation is as follows. Begin from the current platoon position. If there is no path, compute offsets backward from the heading of the platoon. If there is a path, interpolate backwards from the lead position for a fixed distance along the path. If the path bends, interpolate the remaining length along the new segment. The interpolation position gives the drive-to position for the next vehicle. This continues for all vehicles in the platoon. Note that if we start from the beginning of the path, there are no segments to interpolate along. In order to simplify the code, an additional segment is added to the beginning of the path along the original heading of the platoon.

Obstacle avoidance

The above works quite well but doesn't take obstacles into account. See figure 7.1, which shows a formation passing by a terrain feature – the drive-to positions on one side actually pass within the obstacle. Even worse is a narrow spot, where drive-to positions on both sides pass within the obstacle (figure 7.2). The worst case is where the platoon passes by a narrow sliver of an obstacle, which can strip off one of the outer vehicles and destroy the formation (figure 7.3).

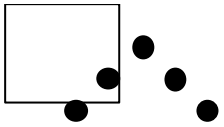

Figure 7.1: Formation collision with one obstacle

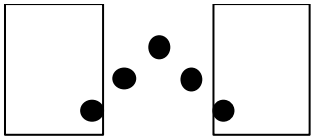

Figure 7.2: Formation collision in narrow spot

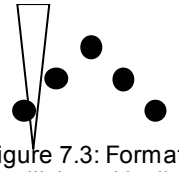

Figure 7.3: Formation collision with sliver

How to compensate for this? Well, we know that the path computed by the main platoon pathfinding algorithm is always safe. So, if the line from the lead vehicle position to another formation position crosses an obstacle or a container boundary, the drive-to position sent to the vehicle is pushed away from the obstacles towards the line defined by the path, i.e. it is interpolated towards a column position. When the formation position moves out of the obstacle the drive-to position is interpolated back to the original formation position.

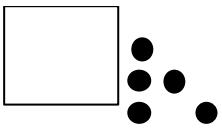

Figure 7.4: Formation modified with one obstacle

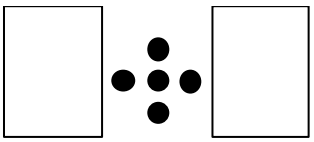

Figure 7.5: Formation modified in narrow spot

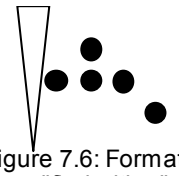

Figure 7.6: Formation modified with sliver

5 Vehicle-Level Pathfinding

The above works reasonably well for computing positions for units that can turn quickly and move in any direction. The vehicles that we dealt with in Force 21 were not so flexible. Helicopters and tanks take time to turn and accelerate. Wheeled vehicles rotate in place to any facing – they can either turn left or right in a limited radius. To turn around, they either need to drive in a large arc, or back up while turning in one direction, and then move forward, turning in the opposite direction (the familiar three-point turn).

In addition, vehicles need to avoid each other, and vehicles in other platoons. While collisions can't always be avoided, it makes the platoons and the vehicles in them look more intelligent if they behave like they're trying to avoid them.

To meet these goals, some crew behaviors based on Craig Reynolds' steering behaviors [Reynolds] were created.

Drive-to position

The first task is to get the vehicles to follow and arrive at the drive-to points described above under formations. This is done through a modification of the seek behavior described by Reynolds. First, the vector from the current position of the vehicle to the drive-to point is computed. This gives the base velocity and steering direction. The length of the vector is the base speed. If this is less than a certain value (2.5 meters), the vehicle has arrived and no change in steering or velocity is computed. If it is less than the maximum speed of the vehicle, the vehicle needs to brake, and the speed is halved. Otherwise it is clamped to the maximum speed. The velocity vector is scaled to this value and sent by the crew to the vehicle as a steering command.

The desired heading of the vehicle is determined by the platoon. As previously mentioned, as the platoon moves along its path it turns to follow the path, and may even turn to face backwards due to the whim of the user. In the ideal case, the vehicles in that platoon should face in the same direction, but as discussed below, this does not always happen.

Desired heading and velocity

As discussed above, the AI sends the vehicle a desired heading and velocity. How the vehicle responds to that depended on the type of vehicle.

After some experimentation, it was determined that the best way to break this down was to compare three vectors –the desired velocity, the desired heading, and the current heading. This boils down to four cases. In this discussion, "the same direction" means that they fall within 180 degrees of each other. The four cases can be made clearer by presenting them in terms of a goal, i.e. a seek behavior.

They are all pointing approximately the same direction

In this first case, we want to reach a goal that is in front of us, and end up at approximately the same heading that we have now. This can be met by driving forward and turning gradually toward the goal. This is true for all three vehicle types. Once the goal is reached, the tracked and rotary wing vehicles can stop and turn towards the proper heading – the wheeled vehicles can't turn in place so just stop there.

Desired velocity and heading are pointing the same direction, but current heading is not

In the second case, we want to reach a goal that is behind us, and end up facing the opposite direction from where we are now. With a tracked or rotary wing vehicle, this is met by turning in place until the conditions meet those for case one, at which time it moves forward and turns towards the goal. A wheeled vehicle cannot turn in place, however, so it needs to back up while turning until conditions for the first case are met. The direction of turn is determined by the difference between the two headings – if the difference is positive, the wheels are turned to the right, otherwise they are turned to the left.

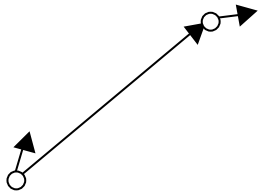

Figure 8.1: goal in front

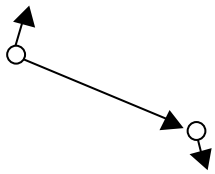

Figure 8.2: goal behind, facing away

Desired heading and current heading are pointing the same direction, but desired velocity is not

In the third case, we want to reach a goal behind us but with similar heading. Most of the time this can be achieved by driving backwards and turning gradually to the goal. For long distances, however, this is not smart behavior, particularly with tracked and rotary wing vehicles. In this case, they will back up while turning around. Once they are facing the goal, this switches to case four, below. Wheeled vehicles will back up and turn to face the goal as in case two, at which point they will also switch to case four.

Desired velocity and current heading are pointing the same direction, but desired heading is not

In the fourth case, we want to reach a goal ahead of us, but turn around when we get there. For tracked and rotary wing vehicles, again, this is easy. We drive there normally, and turn around upon arrival. Wheeled vehicles, as above, will head to the goal, but cannot turn once they get there so will end up facing the wrong direction. This was accepted as a compromise considering the difficulties of steering them *and* maintaining formation. In compensation, their weapons were set so that they could fire in an arc of 360 degrees, so even though they were facing backwards, they could defend themselves.

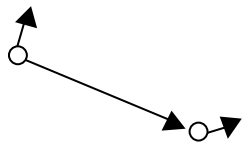

Figure 8.3: goal behind, facing similar

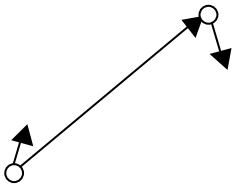

Figure 8.4: goal in front, facing back

Check for bad position

This generally works well except in certain cases, where an obstacle falls between a drive-to point and its corresponding vehicle. This can happen when a platoon is making a sharp turn around a building and the vehicles are just a little too far behind (Figure 9). It not desirable for the vehicle to hit the building/obstacle for aesthetic and practical reasons (the vehicle may get stuck), so this case must be detected and handled. This is detected by doing periodic checks.

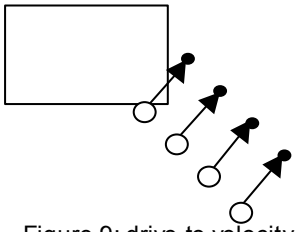

Figure 9: drive-to velocity crosses obstacle

If the vehicle is a beyond a certain distance from the drive-to point ($\sqrt{3}$*maxSpeed) then check to see if the line from the vehicle to the drive-to point crosses a pathfinding boundary. If so, compute a path for just that vehicle to the drive-to point, and use the path to compute steering direction. The path is followed until almost the end, and then normal driving begins again. Originally the path was followed for only a few nodes so that the vehicle could return to normal driving as soon as possible and not hold up the platoon, but this caused problems in certain rare cases.

Avoid other vehicles

The previous two behaviors provide means for formation following and static object avoidance. Avoiding dynamic objects – i.e. other vehicles – is the last step. As with driving to position, this is based on Reynolds' steering behaviors.

The idea is to predict where other vehicles will pass by the current vehicle in the future, based on current positions and velocities, and steer to avoid future collisions. A full discussion can be found in [Reynolds] – in brief the algorithm is as follows:

```
For all vehicles within a certain radius
     Compute time of nearest approach
     If less than minimum time
          If the two vehicles collide at that time
               Store minimum time and vehicle ID

If there is a possible collision vehicle
     Turn left or right (sharply if directly in front, less if
          farther to the side)
     Slow down (more if vehicle is closer, less if far away)
```

Certain modifications were made for our algorithm. First, the velocity used for the current vehicle is the average of the current velocity and the desired steering velocity (as computed above). This was found to be a better predictor for the future velocity of the vehicle.

Second, Reynolds' algorithm used spheres to detect collisions. Since most of the vehicles are essentially rectilinear this was not granular enough and led to behaviors such as a vehicle steering to avoid one vehicle that it clearly will pass and not avoiding a second vehicle which is slightly behind the first. Instead, spheres were used to cull out cases, and swept spheres (a cylinder capped by two half spheres) were used as the final determiner for possible collision.

Third, if the drive-to point for the current vehicle is too close to the collision vehicle, the current vehicle stops. This handles cases where two platoons have driven to nearly the same destination and are intertwined. Otherwise, two vehicles may endlessly circle each other, trying to get to drive-to points that are very close together but also constantly avoiding each other.

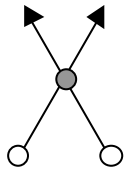

Figure 10: special collision avoidance case

Fourth, if two vehicles are travelling at approximately the same speed and heading but slightly towards each other (see Figure 10), they will either constantly collide side-by-side (if the drive-to impetus is given more weight), or will avoid each other but never get to their destination (if the collision avoidance is given more weight). Our solution is to check for the case and have the one on the right speed up slightly so the other can pass behind.

Finally, the drive-to and collision avoidance velocities need to be combined into the final steering velocity. This is done with the following formula

```
t = (10.0f - minTime)/5.0f;
if (t > 1.0f)
     t = 1.0f;
steering.x += (t*lateralAvoidance.x + forwardAvoidance.x);
steering.y += (t*lateralAvoidance.y + forwardAvoidance.y);
```

where `steering` is the original drive-to velocity, `lateralAvoidance` is avoiding left-right and `forwardAvoidance` is braking. This produces the desired behavior where the vehicle will turn slightly at a distance but more and more as it approaches collision.

One question may arise – why didn't we have the platoons avoid each other as well? This possibility came up, but was rejected because of narrow canyons and bridges in the game. One platoon could place itself in the middle of one of these narrow regions and block other platoons from passing. Providing only vehicle avoidance allowed one platoon to pass through another one, and avoid such issues.

6 Conclusions

We have presented a system for high-level and low-level pathfinding, using visibility graphs, vector offsets for formations, and steering behaviors for vehicles. The algorithms cover roads,

bridges, swamps, formations, and natural vehicle movement in a physically-simulated environment. While in general we are quite pleased with the results, there are some lessons to be learned, and things that we would definitely do differently the next time.

Write a tool for testing pathfinding.

Jon Owen created a tool for testing and maintaining the basic visibility graph algorithm. Without it, the process of creating the road-following and swamp-avoiding algorithms would have been prohibitively difficult. As it was, the remaining parts of the algorithm were quite difficult to debug. Code would be created, and tested with limited test cases, and then passed on to quality assurance to be further tested. A 2D tool for all of the pathfinding would have been extremely useful and would have probably saved weeks of head scratching and keyboard-pounding.

Early save game functionality is good.

On the other hand, having save game functionality in early allowed the testers to provide lots of test cases, particularly instances where vehicles got stuck. Without that the task would have been impossible. As it was, save game was not maintained well and was under a constant state of repair. If one individual had been responsible for it might have remained in better shape and provided even more data.

Use visibility graphs for large-scale movement, A* for local pathfinding.

Visibility graphs work well for moving objects long distances across a map which has two areas – places you can go, and places you can't. They are not so good for maps with rolling terrain where vehicles have variable speeds depending where they are. A*, on the other hand, is good at such maps, but not so good for long distances. One potential solution is to create a high-level visibility graph which covers those terrain features which are impassible, and then use A* to compute a new path along each edge in the visibility graph path, which takes advantage of the local terrain. This may give the advantages of both algorithm – it may also be computationally too expensive. Further examination of this technique may occur in later games.

Better platoon-level pathfinding for helicopters.

Ignoring variability in terrain was particularly noticeable with rotary wings. In general they can go almost anywhere, but some places are better than other. For example they will want to follow canyons and other low-lying terrain so as to avoid notice, but will also want to be able to briefly fly over a hill to a new area of terrain. Marking some hilltops with swamps might have solved this problem in the Force 21 case, but again, a hybrid visibility graph and A* algorithm would probably have been better.

Have platoons slow down slightly when passing obstacles.

After shipping, some rare problems with vehicles getting lost or stuck were still noted when platoons pass by some buildings on certain levels. This might have been handled better if the platoon slowed down to allow vehicles to carefully move around the buildings using the morphing formations instead of calculating their own paths.

7 Acknowledgments

Many thanks to John O'Brien for his good advice and calm assurance during a critical point in the project.

Thanks also to Gary Stelmack, Jeff Friedlander and Derek Earwood for never accepting less than the best in quality (despite pleas otherwise).

References

Bondy, J. A. and U.S.R. Murty, *Graph Theory with Applications.* North-Holland, 1976.

Even, Shimon, *Graph Algorithms*, Computer Science Press, 1979.

O'Rourke, Joseph, *Computational Geometry in C*, Cambridge University Press, 1994.

Pottinger, Dave, "Coordinated Unit Movement," *Game Developer*, pp. 42-51, January 1999.

Pottinger, Dave, "Implementing Coordinated Movement," *Game Developer*, pp. 48-58, February 1999.

Reynolds, Craig, "Steering Behaviors for Autonomous Creatures," in Game Developer Conference Proceedings, pp. 763-782, 1999.

Stout, Brian, "Smart Moves: Intelligent Pathfinding," *Game*, pp. 28-35 *Developer*, October/November 1996.

Vincke, Sven, "Real-Time Pathfinding for Multiple Objects," *Game Developer*, pp. 36-44, June 1997.