Formation Movement for Real-Time Strategy Games

Andrew T. Christensen, Squirrel Eiserloh

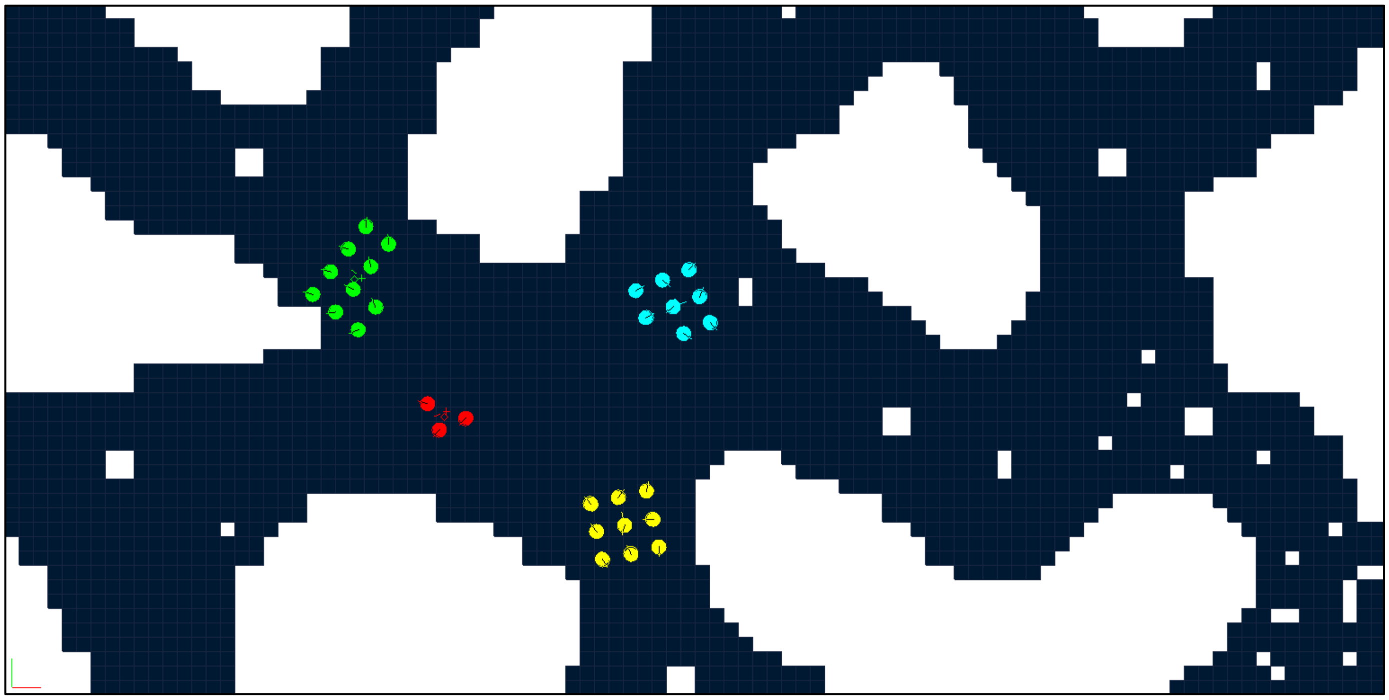

Figure 1: The test bed application, with a sample map and several unit formations. Units are represented by circles. Impassable areas are drawn in solid white. Each formation draws in a different color.

***Abstract*— Maintaining unit formations in an RTS game is an interesting and challenging AI problem. The goal of this research project was to solve the problem as completely and generically as possible, so that other developers may be able to form conclusions about how best to implement such a system for their own projects. Whereas much of the extant research focuses on one specific area of the larger problem, we will attempt to provide developers with a concrete example of such a system in action as well as an exploration of the problem space.**

***Index Terms*—Intelligence and AI in Games, Computer Simulation, Real-Time Systems, Cooperative Systems, Pathing, Formation, Coordinated Movement**

Manuscript received April 10, 2014.

Andrew T. Christensen is a graduate of The Guildhall at SMU, Plano, TX 75024 USA (phone: 817-821-2638; e-mail: andrew@andrewtc.com).

Squirrel Eiserloh is a lecturer at The Guildhall at SMU, Plano, TX 75024 USA (e-mail: beiserloh@smu.edu).

I. Introduction

REAL-TIME STRATEGY (RTS) games are complex battle simulations in which each player takes command of an army of soldiers (called *units*) and must defeat all opposing enemy forces. RTS games mimic conventional warfare by giving the player incomplete knowledge of the battlefield and limited resources. This scenario necessitates scouting the terrain, managing an economy, and coordinating strikes on enemy forces while minimizing casualties. The extent to which players must micromanage individual units varies from game to game, but all RTS games provide players with some mechanism for controlling the behavior of their respective units on the battlefield.

Because unit control comprises a vital part of the user experience of an RTS game, developers typically devote a large amount of time and resources to making unit control as effortless and intuitive as possible. To achieve this, RTS games such as *Starcraft* and *Supreme Commander* provide high-level controls for ordering units to a specific location on the map without specifying the exact path that each unit will take to get there. Players are typically able to select multiple units with the mouse and then click at a location on the map to order all selected units to that location. In order to make this possible, the game typically employs a pathfinding strategy of some kind to determine how each unit will get to the destination. Once the pathfinding step is complete, the result is passed to a unit-level AI that is responsible for managing the movement of each unit in such a way that it eventually reaches the destination. Because the specifics of unit movement vary from game to game, there is no universally accepted way to implement this system that is applicable to all games in the genre. On the contrary, unit movement is an area under constant research, and no two games address the problem in exactly the same way.

Some games, such as *Age of Empires* and *Supreme Commander 2*, enforce additional constraints on unit movement by keeping units in organized formations. This simulates the behavior of conventional armies, which traditionally march together in organized ranks across the battlefield. Games that keep units in formation make use of formation-level AI that coordinates the movement of individual units within the larger group. A good formation-level AI is able to efficiently move units into formation and keep them organized, but is flexible enough to allow units to break ranks as necessary. For example, it is often more efficient for units to split up to go around an obstacle and regroup on the other side (as shown in **Figure 2**), rather than moving around the obstacle in formation. Units should also be allowed to disperse when engaging enemy units at close range (as shown in **Figure 3**). In this scenario, dispersing allows the entire formation to surround and attack the enemy, which could result in fewer casualties than staying in formation. Building a formation-level AI that handles edge cases such as these is non-trivial, and is another area of constant research.

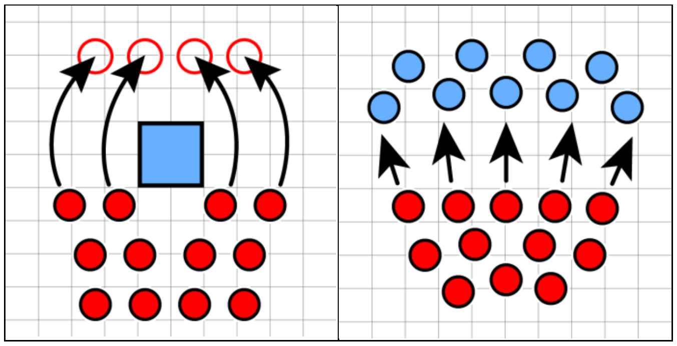

Figure 2 (left): A formation (shown in red) encountering an obstacle (shown in blue). The black arrows demonstrate how the formation-level AI might move units to avoid the obstacle, by having units disperse to go around the obstacle and regroup on the other side. **Figure 3 (right):** A formation (shown in red) engaging an enemy force (shown in blue). In this situation, the formation-level AI should allow units to fan out and get as close to the enemy as possible, rather than keeping them in organized ranks.

The goal of this research project was to explore various methods of achieving formation-based movement and assess their benefits and drawbacks. We were able to create a two-tiered AI that is capable of managing units at both an individual and a group level. It consists of a formation-level AI, which determines where each unit should stand relative to each other to maintain cohesion, and a unit-level AI, which moves individual units into formation and out of the way of obstacles. Both tiers work in tandem to steer each formation to its goal as quickly and efficiently as possible.

II. Research Review

Real-time coordinated movement is a broad topic with wide applications in robotics, games, and the defense industry. While there is no universally accepted technique for implementing coordinated movement, much of the current research draws directly or indirectly upon Craig Reynolds' work on the subject of *flocking* behaviors. [1] Reynolds demonstrated how it is possible to simulate seemingly complex group behaviors by enforcing only a few simple constraints on individual movement. In Reynolds' simulation, each individual within the larger group (called a *boid*) constantly steers in the direction that best satisfies three competing constraints: *separation* (maintaining a minimum distance between units), *alignment* (steering in the average direction of the entire group), and *cohesion* (steering toward the center of mass of nearby boids). [1] Enforcing these constraints produces emergent, organic behavior that mimics a flock of birds or a school of fish. Remarkably, maintaining flocking behaviors requires relatively little processing time, since the flocking constraints are resolved on an individual basis without any sort of group coordination.

While flocking behaviors are capable of producing natural-looking formation movement, flocking alone is not adequate in situations where units are likely to collide with obstacles or with each other. In most RTS games, the map is littered with obstacles and impassable terrain that units must successfully navigate in order to reach their destination. RTS games usually impose the additional constraint that units cannot move through each other. For these games, the unit- and/or formation-level AI must make use of pathfinding in order to ensure that units can traverse the map without being impeded by obstacles. The most widely used search algorithm for unit pathfinding is A-star. Since the algorithm is computationally expensive, however, using it to find paths for individual units is usually cost-prohibitive. RTS developers commonly alleviate this problem by finding ways to reduce the frequency with which units search the map. For example, the formation-level AI in *Age of Empires* mitigated the problem by choosing one unit out of each formation (called the "commander") to handle pathfinding. [2] As necessary, each commander would perform searches for the entire formation and all other units in the group would follow it to the destination. [2]

Recent RTS titles, including *Planetary Annihilation* and *Supreme Commander 2*, use *flowfields* to perform optimized pathfinding for large numbers of units. A flowfield is a type of tile-based vector field (i.e. grid of directions) whose vectors represent directions through the map that eventually lead to a common goal location. Flowfields are generated by using a flood fill algorithm, similar to Djikstra's, that expands outward from the goal location, eventually visiting all tiles that are reachable from the goal. As with Djikstra's algorithm, each tile is expanded in order of cost (i.e. the total cost of entering all tiles between the current tile and the destination). For each tile, the algorithm records the direction to the adjacent tile with the lowest cost that has already been expanded (and is therefore closer to the goal). The end result is a map of vectors that "flow" around obstacles to the goal location. (See **Figure 4** and **Figure 5**.) Once the flowfield has been generated, all units headed to the goal location can look at the flowfield data to determine how to get closer to the goal. Each unit simply retrieves the flowfield vector for its current position every frame and moves in the direction that the vector is pointing, which will eventually push the unit into another flowfield tile. In this manner, units can follow the directions through the map to the goal location without having to find a path to the goal (as shown in **Figure 5**). Because A-star pathfinding is expensive, it is faster to generate a single flowfield for large numbers of units than having units request paths to the goal individually.

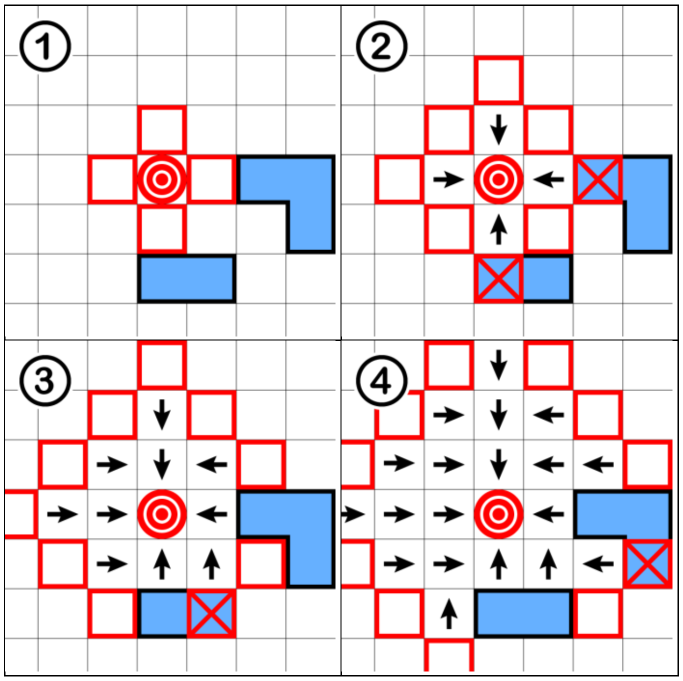

Figure 4: An example of how flowfield generation works. The algorithm starts at the goal (represented by the red target) and progressively expands all adjacent tiles in order of their total cost to reach the goal. For each step of the algorithm, the red outlines above designate the tiles that will be expanded in the next step. Impassable tiles (shown in blue) are not expanded (denoted by the red X's). As each tile is expanded, the algorithm stores a vector pointing toward the adjacent tile with the lowest goal cost (shown above as black arrows). The algorithm will eventually visit all tiles that are reachable from the goal.

Researchers at the University of Washington, seeking to optimize pathfinding for real-time crowd simulations, were the first to conduct extensive research on the subject. [3] The researchers used flowfields to support hundreds of crowd agents moving independently toward different goals at interactive rates. [3] Elijah Emerson subsequently drew upon this research to develop a flowfield-based group pathfinding system for *Supreme Commander 2* that pioneered the use of flowfields in real-time strategy games.

One downside to using flowfields for pathfinding is that

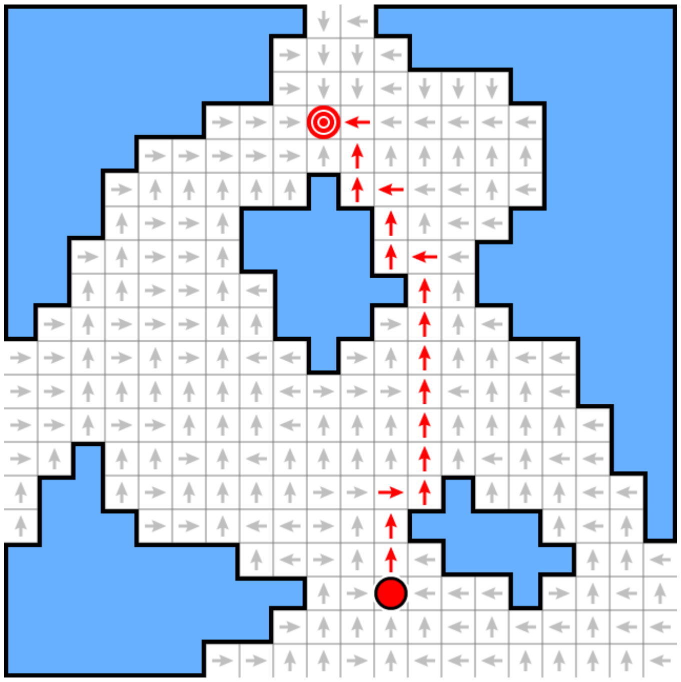

Figure 5: A sample flowfield. Each tile stores a direction to an adjacent tile that is closer to the goal (shown in gray). Units can find a path from any tile to the destination by starting at any tile following the directions in the map (shown in red).

each flowfield is only valid for units travelling toward a common, stationary goal location. For an RTS, this implies that the game must generate a new flowfield dynamically every time the player issues a new movement order. Because flowfields typically record information for the entire map, generating a new flowfield can be prohibitively slow for large maps. [4] To address this issue, Emerson optimized the flowfield generation for *Supreme Commander 2* by using a multi-tiered pathfinding approach to cull areas of the flowfield that did not need to be generated. His solution was to divide the map into a square grid of sectors, each of which was divided into smaller map tiles. Each sector stored detailed adjacency information that could be used to test if two sectors were connected to each other. Instead of generating a flowfield for the entire map, the game would first run an A-star search through the larger sector map to determine which sectors units would pass through while en route to the goal. The game would then generate flowfield data only for these sectors (i.e. only the sections of the map where the flowfield data would be relevant). [4]

While pathfinding helps units avoid static obstacles on the map, pathfinding alone cannot prevent units from colliding with moving obstacles (such as other units). This is especially important for units in the same formation, since they tend to move in close proximity to each other and are therefore likely to collide. RTS games typically employ a variety of collision avoidance techniques to ensure that units can avoid dynamic obstructions. A common technique involves using a *potential field*, a type of heat map that records information about which areas of the map are the most "congested" (i.e. full of moving obstacles). Each tile on the influence map contains a value between 0.0 and 1.0 that represents how close the tile is to nearby obstacles. Each frame, moving obstacles and units increase the value of all nearby tiles to indicate that these tiles are congested. Pathfinding searches can then incorporate the values in the influence map into the entry cost of tiles on the map in order to bias the search away from areas of heavy congestion.

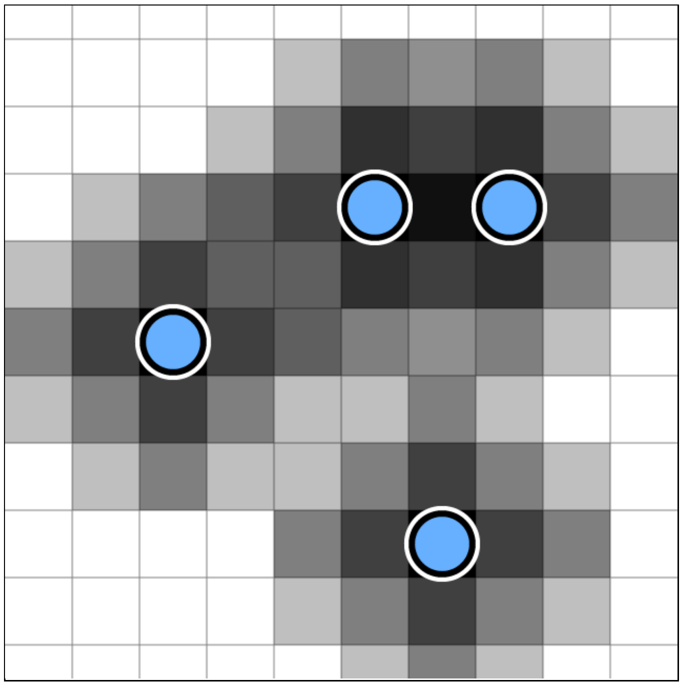
Figure 6: An example potential field. The grayscale color in each tile represents the value at that tile. Values closer to 0.0 are lighter, while values closer to 1.0 are darker. Each unit increases the value of surrounding tiles.

Hägelback demonstrated how potential fields could be used to allow units to quickly find optimal places to stand in their immediate surroundings. [5] In his solution, each unit runs a greedy search through the influence map every frame to determine if any nearby tiles are less congested than its own. If so, the unit will attempt to move into the surrounding tile with the least congestion. Because the tiles surrounding other units have higher values on the influence map, units naturally spread out from each other and move out of the way of obstacles when necessary. [5]

The collision avoidance process becomes is much more difficult for units with acceleration and angular velocity. Introducing these factors implies that units cannot turn in place, and must move forward in order to turn around. To handle this, Dave Pottinger, the AI developer for *Age of Empires*, created a collision avoidance system that calculates a trajectory for each unit and tests for overlap with other unit trajectories in order to predict future collisions. [6] To resolve these situations, a formation-level AI senses future collisions between units and decides the priority of each unit involved in the collision. The higher-level AI lets the unit with the highest priority pass while ordering all other units to stop or slow down in order to avoid the collision. After the higher priority unit has passed, the AI repeats the process until all units have passed each other [7]. One advantage of his design is that it is able to handle *stacked canyon* scenarios easily. These are situations in which multiple units in a narrow corridor must all move out of the way in order to allow some other unit to exit [2]. He shows how defining strict passing rules for units allows the AI to solve this problem without writing code to specifically address this edge case [2].

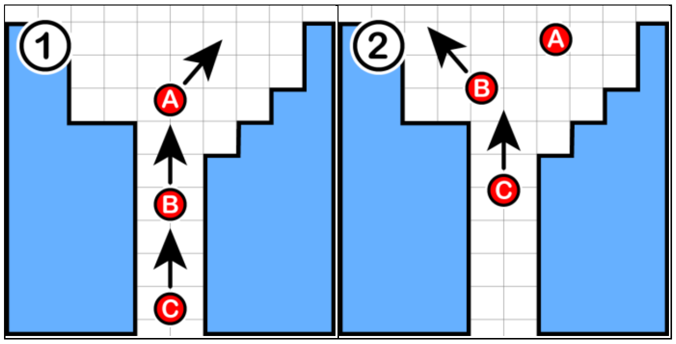

Figure 7: A stacked canyon scenario. In order for unit C to leave the corridor, units A and B must move out of the way first.

There are many different ways to represent the shape and structure of formations. The most common approach is to generate a set of available positions, called *slots*, to which units in the formation are assigned. Each unit moves to its assigned formation slot and follows it as the formation travels across the map. Assigning a specific position to each unit within a squad can cause problems because units tend to block each other from getting to their desired locations. [7] To solve this problem, van der Heijden et al. chose to model formations dynamically using constraints instead of assigning an exact position to each unit. [7] The researchers defined general rules for where units should stand in a larger formation and let each unit determine an optimal position for itself that best satisfied these constraints [7]. An alternate method, described by Clodéric Mars, tries to prevent collisions between units moving into formation by sorting each unit by its position before assigning it to a slot. [8] His algorithm first sorts units by their vertical distance to the formation center. It then divides the formation slots into rows and assigns units to each row in sorted order, with the highest units filling the top row and the lowest filling the bottom row (as illustrated in Figure 8). After each unit has been assigned to a row, the algorithm numbers each slot from left to right by its order within the row. The units in each row are subsequently sorted by their horizontal position and assigned a number from left to right in a similar fashion. Finally, each unit is assigned to the formation slot in its current row with a number that corresponds to its own (e.g. unit 1 would be assigned to slot 1, followed by unit 2 in slot 2, and so forth).

This paper draws upon previous research to come up with a list of suggested approaches for implementing formation movement in a modern RTS. The description of each approach is comprehensive enough to give guidance on overcoming many of the common hurdles that developers face when designing such a system. Specifically, this paper addresses issues related to building and maintaining formations, group pathfinding, handling unit collisions, and handling edge cases that arise when moving through complex terrain.

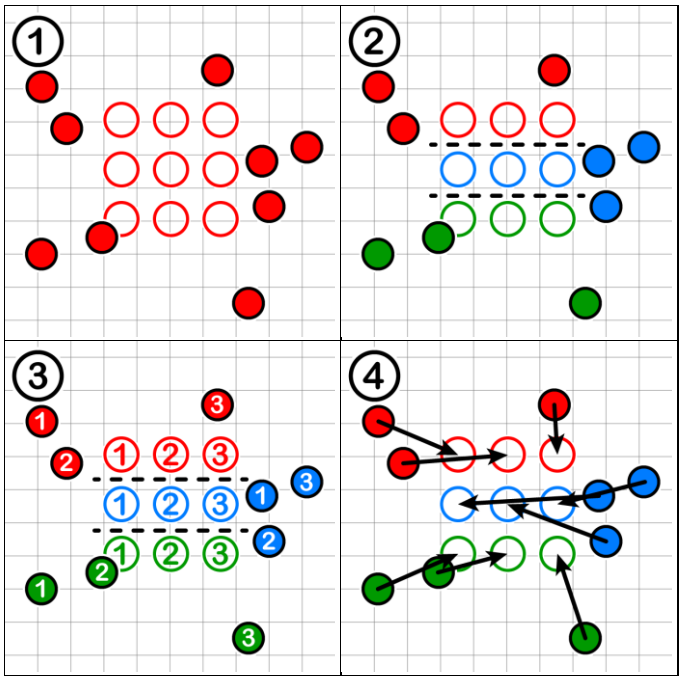

Figure 8: Clodéric's method for resolving collisions between units as they move into formation. **Top left:** The situation before units have been assigned a slot. Formation slots are indicated by hollow circles, while units are indicated by solid ones. **Top right:** The formation slots are divided into rows. Units are assigned to each row from the top down, in order of their height. **Bottom left:** The slots in each row are numbered from left to right by their position within the row. Units are sorted by their horizontal position and assigned a number from left to right. **Bottom right:** Each unit is assigned to the slot in its current row with a matching number.

III. Methodology

Building a formation movement system that works well is a broad and open-ended task. Because development resources are limited, developers must choose carefully which problem areas to address. The requirements for formation movement depend largely on the rules of the game and the expected behavior of units. For our artifact, we chose to prioritize the visual quality of unit movement and pathfinding efficiency over collision avoidance. Our primary goal was to create an AI that was capable of keeping formations organized and effectively leading them to their destinations without sacrificing frame rate.

A. Simulation

Our test bed application consists of a tile-based map that contains multiple units that the user is able to control. We chose to represent the map as a grid of square tiles that are marked either passable or impassable. Passable tiles allow units to enter them, while impassable tiles are solid and collide with units. For the sake of simplicity, we built our movement system with the assumption that the passible/impassable status of each tile would not change during the course of the simulation. With regard to pathfinding, each passable tile is considered to be connected to adjacent tiles in each of the four cardinal directions—north, south, east and west. Units, however, are not constrained by the adjacency of tiles, and may move freely through passable areas in any direction. Units are roughly the same size as a single tile, but can occupy more than one tile at any given point in time.

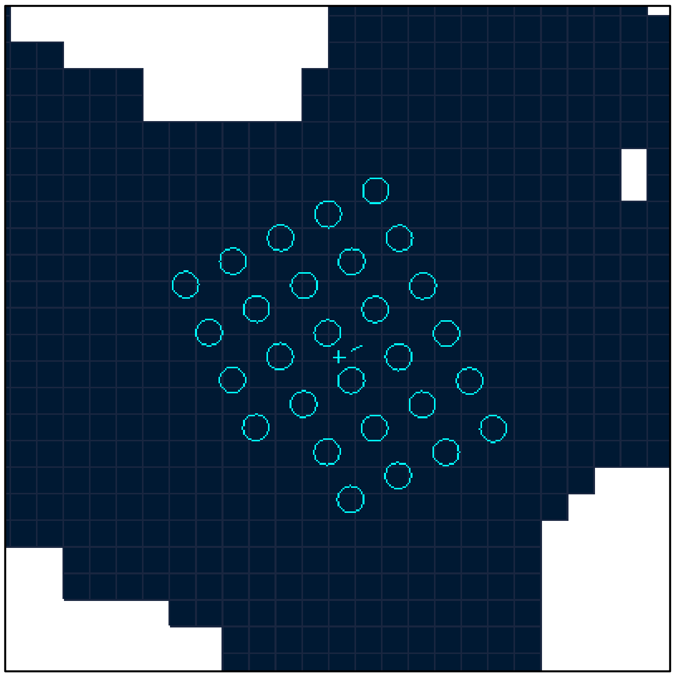

Figure 9: A large formation within the test bed application. The locus of the formation is drawn as a small cross at the center of the formation. A small line indicates the facing direction of the formation. Formation slots are drawn as hollow circles.

Units are independent actors within the world, each with its own position, velocity, and orientation. These are represented as floating point values, so it is possible for units to exist at virtually any location within the bounds of the map. Since introducing acceleration and angular velocity into the physical simulation would make unit movement much more complex, we chose to simplify the problem by allowing units to change their velocity and direction instantaneously. Thus, units always face the direction in which they are moving, and always attempt to travel at their maximum speed when moving. To keep units from overlapping and prevent collisions between units, the simulation applies a repulsive force to units that come within a certain distance of each other. The repulsive force increases exponentially as units get closer to each other, allowing units to overlap partially but not completely.

B. Formations

We chose to model each formation as a list of ideal positions for each unit within the formation, called *slots*. Although they do not collide with impassable tiles, formations are physical actors within the world, with a position, orientation, and velocity. Formation slots are stored as vector offsets from the central position, or *locus*, of the formation. Hence, the locus acts as an origin point for the formation as a whole (as shown in Figure 9). Changing the orientation of the formation will cause all formation slots to rotate around the locus of the formation to maintain the same offset.

Formations are created and destroyed dynamically as the user issues movement orders to different units. Our test bed allows the user to select multiple units with the mouse and then click a location on the map to issue a new movement order to all selected units. The simulation then creates a new formation and assigns all units in the selection to it. To determine the initial position of the formation, the simulation averages the positions of all selected units to determine the center of mass of the selection. It also calculates the vector from the center of mass to the destination point and normalizes it to find an initial heading for the formation.

C. Pathfinding

We attempted to draw upon current research to build a formation-level AI that would handle pathfinding efficiently for large numbers of units. At a high level, there are two separate layers of pathfinding that work together to achieve this goal. Firstly, each formation stores a goal location that represents where the formation should be after it has fulfilled its move order. The formation attempts to find a path through the map and follow it to the goal location. If unsuccessful, the formation simply moves in a straight line toward the goal. Secondly, individual units try stay as close as possible to their assigned slot in the larger formation. Each unit attempts to move in a straight line path to its slot location each frame. For units that are already close to the formation, this solution works well. However, units that are far away from the formation cannot simply head in the direction of their slot location and hope to reach the formation. These units need to find a path through the map to their slot location and follow it in order to get into formation.

Our first attempt to solve this problem involved using A-star to find paths from each unit to its slot in the formation. Units were able to request paths from the Map as necessary when moving toward their target location. Each path was stored as a list of coordinates for each tile leading to the goal. We chose to delegate pathfinding responsibilities to the Map over having units find paths individually because path requests often take more than one frame to complete. To prevent drops in frame rate while fulfilling path requests, the Map would spread out the pathfinding process for each request over several frames, if necessary. Once the search was complete, the Map would notify the requestor that the path was ready. The unit who requested the path could then make use of the path until it was no longer needed.

Despite our efforts to maintain an interactive framerate by amortizing the cost of each path request, our first solution did not scale well when applied to large numbers of units. In an effort to reduce the cost of pathfinding, we implemented a flowfield-based system that allows for efficient pathfinding for both individual units and the formation itself. A unit that is blocked by terrain from reaching its slot location in a straight line must fall back on some alternative method of getting closer to its slot location. Since each unit ultimately follows its formation to the goal location, it is reasonable to assume that any path through the map that leads to the formation goal location will eventually lead units closer to the formation itself. To take advantage of this, each formation generates a flowfield headed toward its target location when it is created. If units are too far away from the formation, they may rely on the flowfield data for the formation in order to head toward the goal location. These units will eventually be close enough to the goal to take a straight-line path to their slot location without becoming stuck. Generating a flowfield for each formation in this manner has the added benefit of allowing the formation itself to find a path to the goal location without having to run a separate A-star search. Instead, the formation itself also follows the flowfield tiles to its destination.

Another significant benefit of using flowfields is that they can be repurposed for obstacle avoidance. Each frame, units in the formation can check the map tile underneath their slot location to determine if it is passable. If not, then it is likely that the formation is partially covering an obstacle. This is the case in when formations move over areas of impassable terrain. In these situations, units stop moving toward their designated slot location and follow the flowfield instead. This causes units to break off from the formation as necessary to go around obstacles and regroup once the formation is on the other side.

D. Unit Movement

One disadvantage of representing the map as a series of tiles is that following paths through the map directly result in movement that looks unnatural. Because each tile is only connected to adjacent tiles in each of the cardinal directions, units following the path directly will only move in cardinal directions in order to reach the goal. This is inefficient because, in most cases, there exists a straight line path that will take the unit closer to the goal in a shorter amount of time than visiting each tile along the way. To resolve this, once per frame each unit uses the flowfield for its formation to find a tile that is closer to the goal than the tile to which it is currently moving. If such a tile is found, the unit performs a straight-line trace through the map to the tile’s location to see if it can reach the tile directly. If so, the unit will change direction to move toward this location instead of the next tile ahead of it in the flowfield. Each frame, the unit repeats this process with each subsequent tile in the flowfield until it finds a tile that it cannot reach via a straight line or the goal location itself. The unit continues to search subsequent tiles for better straight line paths, as new paths may become available as the unit continues to move closer to the goal. This makes the movement of units more believable and reduces the amount of time it takes for units to travel to their destination.

E. Slot Assignment

Formations in the test bed use a variation on the algorithm described by Cloděric Mars to assign units to formation slots. The formation slots are divided into rows. Each unit is then sorted by its position relative to the locus of the formation in order to determine its optimal slot.

Because the user can select and issue new move orders to units that are already in formation, the formation must be able to handle changes to the number of units assigned to it. Whenever units are removed, the formation must reconfigure all slots to ensure that the formation remains cohesive. It must also destroy unused slots to guarantee that there are always exactly the same number of slots in the formation as there are units.

One final restriction on formation movement is that each formation must move slowly enough that its units do not fall behind. If a formation gets too far ahead of its units, units will not be able to keep up and will have to resort to using the flowfield to reach the destination. To ensure that each formation stays close to its units, once per frame the formation counts the number of units that are “in formation” (i.e. within a small distance of their formation slot). It divides this number by the number of slots to get a value between 0.0 and 1.0 (which we have termed the *cohesion factor* of the formation). To account for units using the flowfield, slots that are currently over impassable tiles do not count toward the number of slots when calculating the cohesion factor. The formation then uses this value as a multiplier for its current movement speed to slow down as necessary to allow units to catch up. When all units are in formation (i.e. very close to their designated slots), the cohesion factor approaches 1.0, and the formation travels at maximum speed. If a significant number of units fall behind, the cohesion factor will approach 0.0, causing the formation to slow down or stop to allow the units to catch up. This technique allows the formation to travel at the fastest possible speed toward the goal when all units are in formation, but gives stragglers ample time to catch up when falling behind.

IV. Conclusion

Maintaining unit formations in an RTS game is an interesting and challenging problem that poses many difficult challenges. While our implementation is very basic, we were able to construct an RTS unit movement system with both unit- and formation-level AI that is capable of moving units from one point on the map to another efficiently. Our solution is able to overcome many of the common pitfalls inherent to unit movement in general, and is able to perform optimized pathfinding for large formations of units. The solutions we have proposed to address these problems are by no means exclusive. Yet, we believe that they are will constitute a useful resource for developers who are approaching the problem for the first time.

References

[1] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics,* vol. 21, no. 4, pp. 25-34, 1987.

[2] D. Pottinger, "Implementing Coordinated Movement," Gamasutra, 29 January 1999. [Online]. Available: http://www.gamasutra.com/view/feature/131721/implementing_coordinated_movement.php. [Accessed 13 September 2013].

[3] A. Treuille, S. Cooper and Z. Popovic, "Continuum Crowds," *ACM Transactions on Graphics,* vol. 25, no. 3, pp. 1160-1168, 2006.

[4] E. Emerson, "Crowd Pathfinding and Steering Using Flow Field Tiles," in *Game AI Pro: Collected Wisdom of Game AI Professionals*, Boca Raton, A K Peters/CRC Press, 2013, pp. 307-323.

[5] J. Hägelback, "Potential-Field Based navigation in StarCraft," in *IEEE Conference on Computational Intelligence and Games*, Granada, 2012.

[6] D. Pottinger, "Coordinated Unit Movement," Gamasutra, 22 January 1999. [Online]. Available: http://www.gamasutra.com/view/feature/131720/coordinated_unit_movement.php. [Accessed 12 September 2013].

[7] M. van der Heijden, S. Bakkes and P. Spronck, "Dynamic Formations in Real-Time Strategy Games," in *IEEE Symposium on Computational Intelligence and Games*, Perth, 2008.

[8] H. Danielsiek, R. Stüer, A. Thom, N. Beume, B. Naujoks and M. Preuss, "Intelligent Moving of Groups in Real-Time Strategy Games," in *IEEE Symposium on Computational Intelligence and Games*, Perth, 2008.

[9] C. Mars, "The Simplest AI Trick in the Book," in *Game Developers Conference*, San Francisco, 2014.