23 Crowd Pathfinding and Steering Using Flow Field Tiles

Elijah Emerson

23.1 Introduction

Crowd pathfinding and steering using flow field tiles is a technique that solves the computational problem of moving hundreds to thousands of individual agents across massive maps. Through the use of *dynamic flow field tiles*, a more modern steering pipeline can be achieved with features such as obstacle avoidance, flocking, dynamic formations, crowd behavior, and support for arbitrary physics forces, all without the heavy CPU burden of repeatedly rebuilding individual paths for each agent. Furthermore, agents move instantly despite path complexity, giving AI and players immediate feedback.

23.2 Motivation

While working on *Supreme Commander 2*, we were given the task of improving movement and pathfinding behavior. As in many games with pathfinding, each unit in *Supreme*

Commander would move along a fixed, one-way A* path. Eventually, units would collide with other units, especially when they were moving in formation or moving into battle. When paths cross and units collide, the existing code would stop the units and wait for the conflict to resolve, rather than rebuilding a new path around the obstacle. This is because rebuilding a path every time there is a collision turns into a compounding problem, especially in large battles, where the new path will likely lead to a second and third collision, causing the game to grind to a halt. This behavior repeats across a thousand units, whose controlling players are all frantically clicking at each other's units, essentially begging for them to clash and collide with each other.

To overcome this path rebuilding problem, all movement was engineered to prefer to stay on the same path, resulting in limited physics, formations, AI, hit reaction, and so on. In this way, the pathfinding was limiting the entire user experience.

Because of *Supreme Commander's* one-track pathfinding solution, players would babysit their units as they moved across the map. They would spend their time watching and clicking, watching and clicking, all to help their units cope with the game's ever changing obstacles and environment.

23.3 World Layout

In the *Supreme Commander 2* engine, the world is broken up into individual sectors containing grid squares, where each grid square is 1 × 1 meter and each sector holds 10 × 10 grid squares. There are also portal windows, where each portal window crosses a sector boundary. Figure 23.1 shows an example.

In Figure 23.1, sectors are connected through pathable portal windows. Portal windows begin and end at walls on either side of sector boundaries. There is one portal for each window side, and each portal center is a node in an N-way graph with edges that connect to pathable, same sector portals.

23.4 The Three Field Types

For each 10 × 10 m grid sector there are three different 10 × 10 m 2D arrays, or fields of data, used by this algorithm. These three field types are *cost fields*, *integration fields*, and *flow fields*. Cost fields store predetermined "path cost" values for each grid square and are used as input when building an integration field. Integration fields store integrated "cost to goal" values per grid location and are used as input when building a flow field. Finally, flow fields contain path goal directions. The following sections go over each field in more detail.

23.4.1 Cost Field

A cost field is an 8-bit field containing cost values in the range 0–255, where 255 is a special case that is used to represent walls, and 1-254 represent the path cost of traversing that grid location. Varying costs can be used to represent slopes or difficult to move through areas, such as swamps. Cost fields have at least a cost of one for each grid location; if there is extra cost associated with that location, then it's added to one.

If a 10 × 10 m sector is clear of all cost, then a global static "clear" cost field filled with ones is referenced instead. In this way, you only spend memory on cost fields that contain

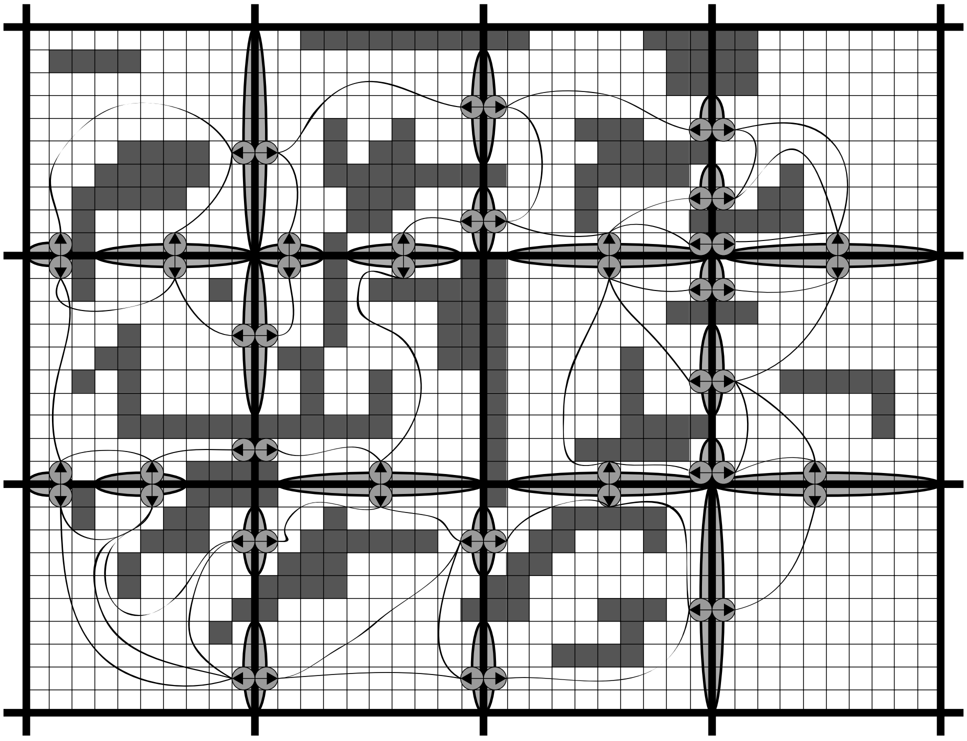

Figure 23.1

An example of the terrain representation used in *Supreme Commander 2*. Each sector is 10 x 10 grid squares with portals connecting sectors.

unique data. In an RTS game, there are a surprising number of clear sectors. In *Supreme Commander 2*, we had roughly 50–70% of the pathable space marked as clear due to widespread areas of open and flat land, lakes, and oceans.

Cost field data was prebuilt by our editor, which converted walls and geometry slope into cost values. Our design team could also visualize this path cost information, as well as make changes to it.

23.4.2 Integration Field

The integration field is a 24-bit field where the first 16 bits is the total integrated cost amount and the second 8 bits are used for integration flags such as "active wave front" and "line of sight." You can optionally spend more memory for better flow results by using a 32-bit float for your integrated cost making it a 40-bit field.

23.4.3 Flow Field

Flow fields are 8-bit fields with the first four bits used as an index into a direction lookup table and the second four bits as flags, such as "pathable" and "has line of sight." The flow field holds all the primary directions and flags used by the agent's steering pipeline for steering around hills and walls to flow toward the path goal.

23.5 Path Requests

Once you have a valid goal position and one or more source positions, you can create a *path request*. The path request will first run A* through the portal node graph. The A* walker starts at the source position, travels through portal nodes, and ends at the goal, thus producing a linked list of "next" portal nodes. This process continues with the next path request source, but this time the portal walker runs "merging" A*, in which the walker prefers to stop and point to a previously traveled portal node to "merge" with previous A* results. With "merging" A* you are more likely to share flow field results and sources are more likely to path closer together, which is the desired behavior when selecting multiple sources to move toward a single goal.

If your A* path to goal is successful, the next step is to walk through your list of next portal nodes and submit a flow field request for each one. At this point you're done with the path request and, because you've only traversed the portal node graph using merging A*, you've used very little CPU.

23.6 The Integrator

We define the *integrator* as the class responsible for taking a single flow field request and, over one or more ticks, building out a single flow field tile. This is achieved by taking the request's cost field data as well as the request's "initial wave front" as input. The initial wave front is a list of goal locations, each having a predetermined integrated cost value.

The integrator takes the initial wave front and integrates it outward using an Eikonal equation [Ki Jeong 08]. Visualize the effect of touching still water, creating a rippling wave moving across the water surface. The Integrator's active wave front behaves similarly in how it moves across the pathable surface while setting larger and larger integrated cost values into the integration field. It repeats this process until the active wave front stops moving by hitting a wall or the sector's boarders. To better understand the integration process, let's go over the Integrator's integration steps.

23.6.1 Integration Step 1: Reset the Integration Field

The integrator's first step is to reset its integration field values and apply the initial goal wave front. If the requested flow field has the final 1×1 goal, then its initial goal wave front is a single 1×1 location with a zero integrated cost value. However, if the flow field request is from a 10×1 or 1×10 portal, then there will be ten goal locations with ten different integrated cost goals.

For higher quality flow results you can integrate at least one flow field ahead in the portal path. Then you can carry over the previously integrated costs as your initial portal window costs instead of using zeros, effectively making the flow across borders seamless. This quality improvement comes at a cost of making flow tiles order dependent, and thus harder to reuse by other path requests.

23.6.2 Integration Step 2: Line Of Sight Pass

If we are integrating from the actual path goal, then we first run a line of sight (LOS) pass. We do this to have the highest quality flow directions near the path goal. When an agent is within the LOS it can ignore the Flow field results altogether and just steer toward the exact goal position. Without the LOS pass, you can have diamond-shaped flow directions around your goal due to the integrator only looking at the four up, down, left, and right neighbors.

It's possible to improve flows around your goal by looking at all eight neighbors during the cost integration pass, but we wouldn't recommend it; marking LOS is cheap and when within the LOS, you get the highest quality path direction possible by ignoring the flow field altogether.

To integrate LOS you have the initial goal wave front integrate out as you normally would, but, instead of comparing the cost field neighbor costs to determine the integrated cost, just increment the wave front cost by one as you move the wave front while flagging the location as "Has Line of Sight." Do this until the wave front hits something with any cost greater than one.

Once we hit something with a cost greater than one, we need to determine if the location is an LOS corner. We do this by looking at the location's neighbors. If one side has a cost greater than one while the other side does not, we have an LOS corner.

For all LOS corners we build out a 2D line starting at the grid square's outer edge position, in a direction away from the goal. Follow this line across the grid using Bresenham's line algorithm, flagging each grid location as "Wave Front Blocked" and putting the location in a second active wave front list to be used later, by the cost integration pass. By marking each location as "Wave Front Blocked" the LOS integration wave front will stop along the line that marks the edge of what is visible by the goal.

You can bring LOS corner lines across sector borders by carrying over the "Has Line of Sight" and "Wave Front Blocked" flags at portal window locations. Then, when you build out the neighbor's integration field, for each portal window location that has the "Wave Front Blocked" flag, consider it an LOS corner to the goal and build out the rest of the line accordingly. This will make the LOS seamless across sector borders.

Continue moving the LOS pass wave front outward until it stops moving by hitting a wall or a location that has the "Wave Front Blocked" flag. Other than the time spent using Bresenham's line algorithm, the LOS first pass is very cheap because it does not look at neighboring cost values. The wave front just sets flags and occasionally detects corners and iterates over a line.

Figure 23.2 shows the results of a LOS pass. Each clear white grid square has been flagged as "Has Line Of Sight." Each LOS corner has a line where each grid square that overlaps that line is flagged as "Wave Front Blocked."

23.6.3 Integration Step 3: Cost Integration Pass

We are now ready for cost field integration. As with the LOS pass, we start with the active wave front list. This active wave front comes from the list of "Wave Front Blocked" locations from the previous LOS pass. In this way we only integrate locations that are not visible from the goal.

We integrate this wave front out until it stops moving by hitting a wall or a sector border. At each grid location we compute the integrated cost by adding the cheapest cost field and integrated cost field's up, down, left, or right neighbors together. Then repeat this Eikonal equation process again and again, moving the wave front outward toward each location's un-integrated, non-walled neighbors.

During integration, look out for overlapping previously integrated results because of small cost differences. To fix this costly behavior, make sure your wave front stops when it hits

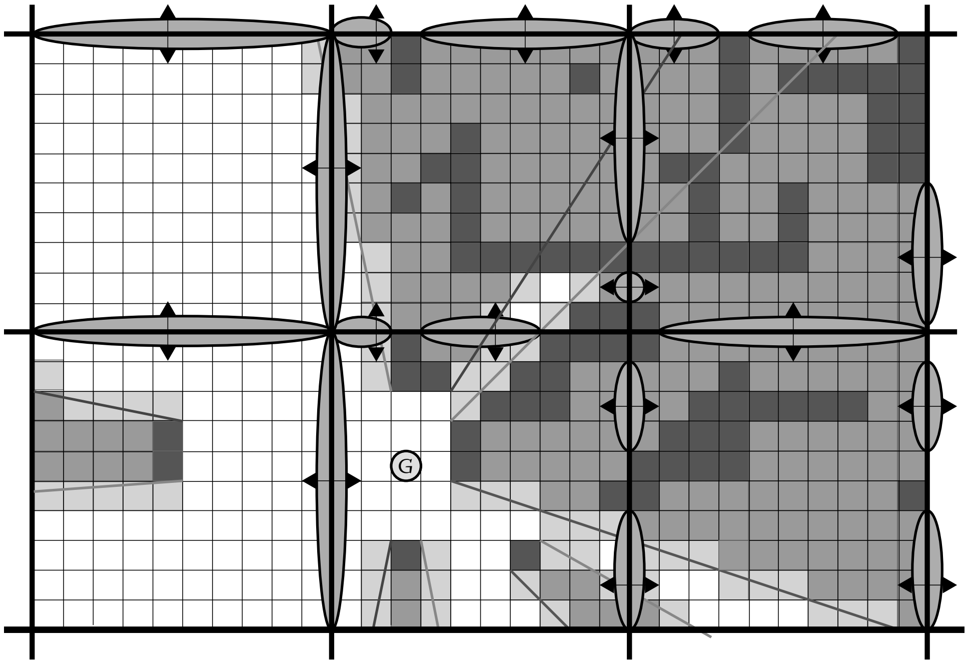

Figure 23.2

The results of an LOS pass.

previously integrated results, unless you really have a significant difference in integrated costs. If you don't do this, you risk having wave fronts bounce back and forth, eating up results when it's not necessary. In other words, if a different path is slightly cheaper to take, then don't bother backtracking across the field just to save that small pathfinding cost difference.

The following is an example of when it's appropriate to overlap previously integrated cost results. Imagine a single path that splits into two paths, where each split path leads to the same goal location. However, one split has a long and costly sand trap, while the other does not. The integration wave front will move away from the goal, split into two, and converge on each other at the beginning of the path. When they meet, the cheaper wave front will overlap the more expensive wave front's results and continue to integrate, backtracking down the expensive path until the cheaper integrated costs do not have a significant difference with the previously integrated costs. This backtracking behavior will have the effect of redirecting the flow field directions away from the sand trap and back toward the cheaper path. This is no different than backtracking in A*; it's just good to point this behavior out as it's more costly when integrating fields.

23.6.4 Integration Step 4: Flow Field Pass

We are now ready to build a flow field from our newly created integration field. This is done by iterating over each integrated cost location and either writing out the LOS flag or comparing all eight NW, N, NE, E, SE, S, SW, W neighbors to determine the "cheapest" direction we should take for that location.

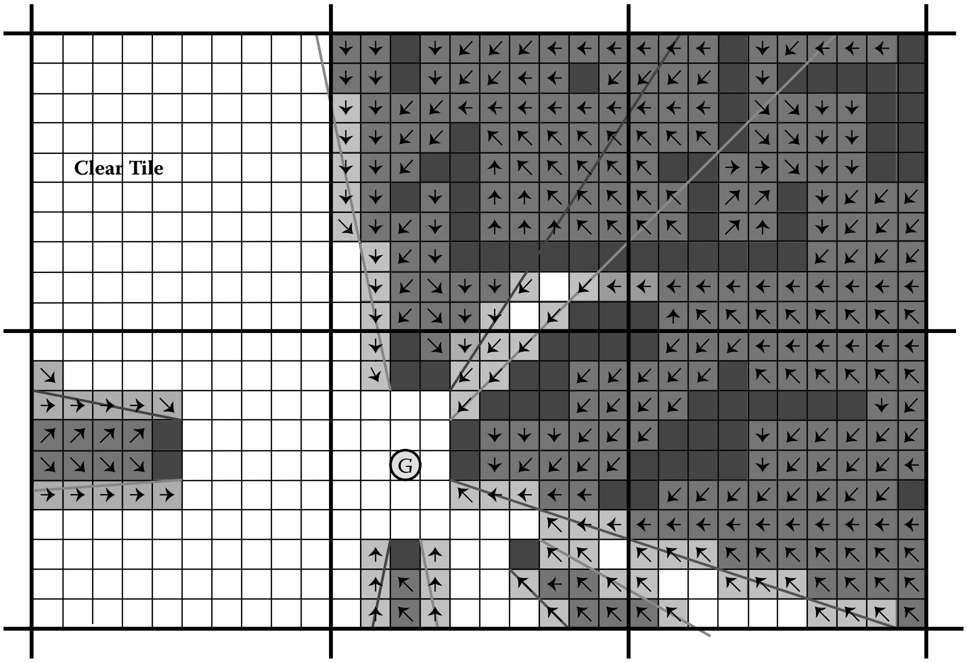

Figure 23.3

The final flow field directions.

Figure 23.3 shows what the final flow field directions look like. Notice that no work was done for locations that have goal LOS or locations within the clear tile. Once the flow field is built out we submit it to the *flow field cache*.

23.7 The Flow Field Cache

The flow field cache contains all of our built flow fields, each with their own unique ID based on the portal window they take you through. In this way, work can be shared across path requests despite having different goals.

If there is a two-way hallway in your map, the odds are pretty good that multiple paths will want the same hallway flow field results. A flow field can also be reference tracked so when there are no more references to it, it can be discarded or put on a timer to be discarded later. You can also prebuild all flow field permutations and store them on disk so that you only need to build the flow fields that have custom LOS goal information.

23.8 Supporting Dynamic Environments and Queries

The whole point of inventing this technique was to better handle the dynamic nature of our game environments in real-time. To that end, we built everything with dynamic change in mind.

We can easily support moving sources by running another "merging" A* across portal nodes if the agent's position moves outside the sectors in the planned path.

We support moving goals by rebuilding the goal's flow field. If the goal crossed a sector boundary, the path's portal nodes are rebuilt behind the scenes. Most of the flow fields requested by the new path will already have been built and will be in the cache, so very few flow fields need to be rebuilt. Once the new path is ready, the agent will seamlessly switch over to it from the old path.

We support changing walls and hills by marking the cost field of the sector that contains them and their associated portals as dirty. Then the portal graph is rebuilt for nodes that are on the borders of the dirty sectors as well as their neighbors. Finally, the paths that were affected by those changes are rebuilt.

All of this is done by marking things dirty and rebuilding them based on a priority queue, where each item in the queue is given a time slice of a fixed number of milliseconds. This allows us to control what, when, and how rebuilding happens over time.

23.9 Cost Stamp Support

Cost stamps represent a custom set of costs values you can "stamp" into the world. In *Supreme Commander 2*, we needed to place buildings down that had custom walls as well as custom pathable areas. The player can essentially paint whole new pathable landscapes by using varying sized structures, including 1×1 grid walls.

Cost stamps record the original cost field values before replacing them with a new set of costs. After placing a cost stamp down, the overlapping sectors would be flagged as dirty and the dynamic graph and path rebuilding process would take care of everything else.

23.10 Source Cost Data

The map editor would build the cost field data from looking at geometry, placing down walls and hills where appropriate. We would run a blur pass to add a cost gradient near walls to improve flow results when going down hallways and around jagged edges.

All cost data was also shown in the editor so that designers could manually add and remove path cost as they saw fit. This was a huge benefit to the design team as they could finally control where and how units moved in their maps.

23.11 Different Movement Types

Each agent in the *Supreme Commander 2* engine has its own movement type. Each movement type has its own cost field data and hence produces its own portal graph. In this way, a land-only tank would have a different path than a hovercraft that can travel over lakes and swamps.

The editor would build out the different cost data for each movement type. To support large units, a special wall cushioning process was run over the map that moved the walls outward. This had the effect of closing off skinny gaps that are too small for large units as well as pushing out wall and mountain sides so large units can't visually overlap them when near.

If the user selected units with different movement types, such as a squadron of jets, a few land-only tanks, some hovercraft, and a super large experimental robot, the game would use the "most restrictive movement type" path for all compatible units before building more paths for the incompatible units.

23.12 Steering with Flow Fields

When agents steer with flow fields, there are some if-else conditions to look out for. For starters, if the agent doesn't have a valid flow field, it should steer to the next portal position. Once the agent has a flow field, it should look for an LOS flag to steer to its goal; otherwise, it should use the specified flow field direction.

When an agent is receiving new flow field directions, we recommend storing off a path direction vector and blending in new flow directions as you cross grid squares. This has the effect of smoothing out the flow field directions as the agent traverses the field.

23.13 Walls and Physics

With flow fields, your pathfinding agents can move in any direction without the high expense of rebuilding their path. Once your agents move in any direction, they are bound to hit a wall or another agent. In the *Supreme Commander 2* engine, agents could push each other around as well as slide along walls using physics.

Having physics in our game allowed for new game play scenarios such as explosions that push back units or super large robots that could push back a hundred tanks. We had a structure that could arbitrarily push or pull units across the map, as well as a large unit that could suck units into a whirlwind, spinning them around and around until they smashed together. These new game play scenarios would not have been possible without the cheaper movement cost associated with using flow field tiles.

23.14 Island Fields

An optional *island field* type can be implemented containing island IDs, where each island ID represents a single pathable island. Imagine the different islands of Hawaii: if you are on one island, you can only drive to locations within the same island.

For each sector you store its island ID. If there is more than one island ID in the sector, then the sector has an island field breaking the IDs down to individual grid locations. With this information you can quickly determine if a path request is valid.

In the *Supreme Commander 2* engine, you can move your mouse over any location on the map and see the mouse icon change from an arrow to a stop sign, indicating that you cannot reach that location. This feature was implemented by retrieving Island IDs at the source and destination locations to see if they match.

23.15 Minimizing CPU Footprint

You can enforce low CPU usage by capping the number of tiles or grid squares you commit to per tick. You can also easily spread out integration work across threads because the Integration Field memory is separate from everything else.

23.16 Future Work

The following is a list of ideas to further improve this technique.

- Support 3D spaces by connecting portal graph nodes across overlapping sectors.
- Pre-process and compress *all* flow field permutations and stream them from disk.
- Add support for arbitrarily sized maps by using a hierarchy of sectors and N-way graphs.
- Build out the flow field using the GPU instead of the CPU [Ki Jeong 07].
- Support multiple goals. Multiple goal flow fields are perfect for zombies chasing heroes.

23.17 Conclusion

Our work on *Supreme Commander 2* shows that it's advantageous to move beyond single path-based solutions and start looking at field-based solutions to support dynamic crowd pathing and steering in RTS games with hundreds to thousands of agents. In this article, we demonstrated how to represent and analyze the pathable terrain to generate flow fields that can drive hundreds of units to a goal. Additionally, we showed that this method is computationally cheap, compared with individual unit pathfinding requests, even in the face of dynamic terrain and queries. Hopefully you can benefit from our experience with the *Supreme Commander 2* engine and continue to expand and refine field-based pathfinding in your next game.

References

[Ki Jeong 07] W. Ki Jeong and R. Whitaker. "A fast Eikonal equation solver for parallel systems." *SIAM Conference on Computational Science and Engineering*, 2007.

[Ki Jeong 08] W. Ki Jeong and R. Whitaker. "A fast iterative method for Eikonal equations." *SIAM Journal on Scientific Computing* 30(5), 2008.