

©Copyright 2008 DigiPen Institute of Technology and DigiPen (USA) Corporation. All rights reserved.

# **CIG-C: A Hierarchical Approach to Continuum Crowds**

BY

Christopher Mitchell Deeb

B.S. Computer Science, the University of Georgia, 2006

## **THESIS**

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
in the graduate studies program  
of DigiPen Institute Of Technology  
Redmond, Washington  
United States of America

Summer  
2008

Thesis Advisor: Xin Li

DigiPen Institute Of Technology  
Graduate study program  
Defense of thesis

The undersigned verify that the final oral defense of the  
master of science thesis of  
has been successfully completed on  
TITLE of thesis:  
Major filed of study:

Christopher Mitchell Deeb

7 / 24 / 08

CIG-C: A Hierarchical Approach to Continuum Crowds

Computer Science

Committee:

\_\_\_\_\_  
Xin Li, Chair

\_\_\_\_\_  
Matt Klassen

\_\_\_\_\_  
Gary Herron

\_\_\_\_\_  
Michael Aristidou

Approved :

\_\_\_\_\_  
Graduate Program Director

\_\_\_\_\_  
date  
Associate Dean

\_\_\_\_\_  
date

\_\_\_\_\_  
Department of Computer Science

\_\_\_\_\_  
date  
Dean

\_\_\_\_\_  
date

The material presented within this document does not necessarily reflect the opinion of  
the Committee, the Graduate Study Program, or DigiPen Institute Of Technology.

INSTITUTE of DigiPen Institute Of Technology  
Program of Master's degree  
Thesis approval

DATE: 7 / 24 / 08

Based on the CANDIDATE'S successful oral defense, it is recommended that the thesis prepared by

Christopher Mitchell Deeb

ENTITLED

CIG-C: A Hierarchical Approach to Continuum Crowds

Be accepted in partial fulfillment of the requirements for the degree of master of computer science from the program of Master's degree at DigiPen Institute Of Technology.

\_\_\_\_\_  
Thesis Advisory Committee Chair

\_\_\_\_\_  
Director of Graduate Study Program

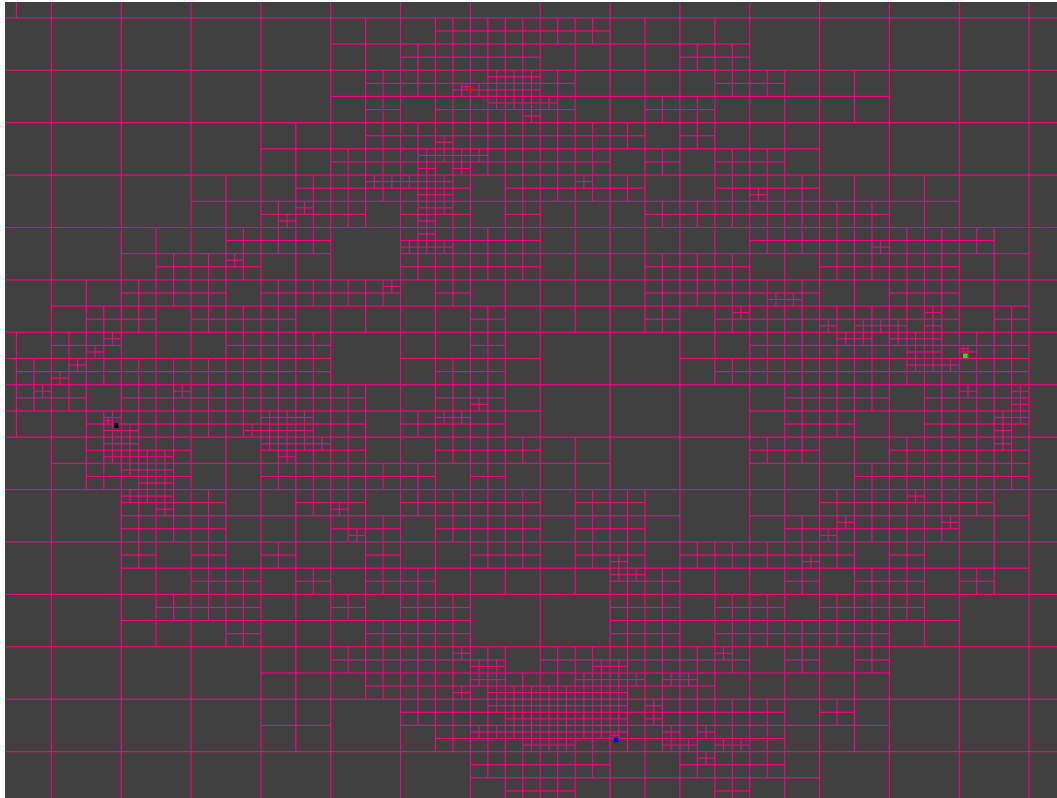
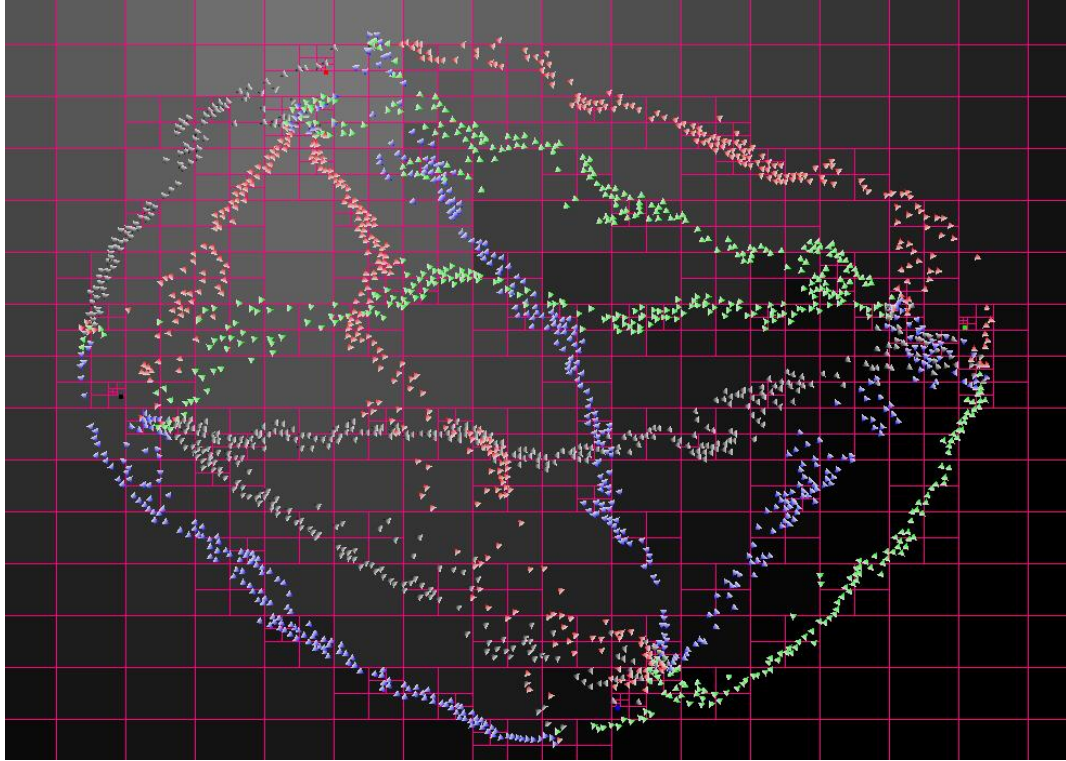
\_\_\_\_\_  
Associate Dean

\_\_\_\_\_  
Dean of Faculty

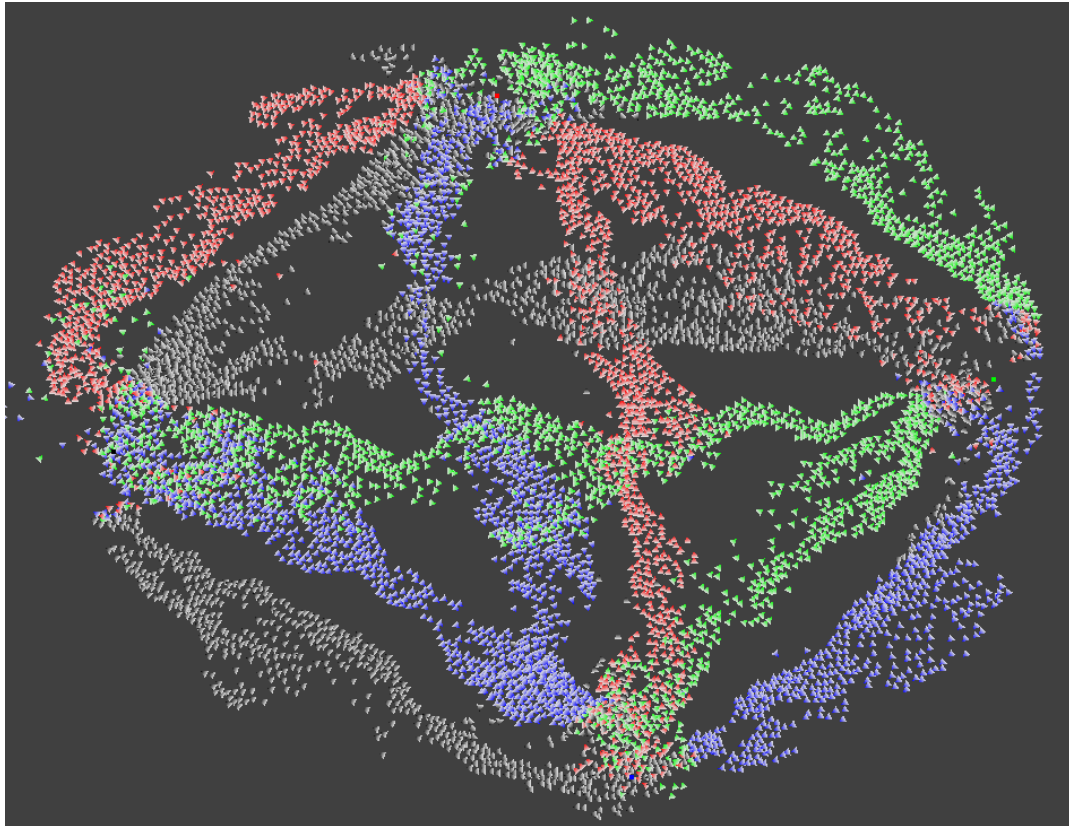
The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute Of Technology.

Abstract.....	9
1 Introduction.....	10
1.1 The Basics.....	10
1.2 Some Uses for Path Planning.....	10
2 Previous Work .....	12
2.1 Flocks, Herds and Boids .....	12
2.2 Dijkstra’s Algorithm.....	14
2.3 The A* Algorithm.....	20
2.4 Fast March .....	21
2.5 Continuum Crowds .....	24
2.5.1 Brief Overview.....	24
2.5.2 The Governing Hypotheses.....	25
2.5.3 Basic Walkthrough.....	30
2.5.4 Results.....	31
3 Our Approach: CIG-C.....	33
3.1 High Concept .....	33
3.2 Significant Differences .....	35
3.3 Creation of the Field - Before the Action .....	37
3.3.1 Tree Structure.....	37
3.3.2 Breakdown of a Single Frame.....	38
3.3.3 The Reasoning Behind the Quad Tree .....	39
3.4 Algorithm Overview: .....	44

3.5 Step One: Frame Start.....	44
3.5.1 Resetting the Field: .....	45
3.5.2 Splattering the Units: .....	45
3.5.3 Pre-Active-List Construction:.....	45
3.5.4 Active List Construction:.....	48
3.5.5 Post-Active-List Construction: .....	51
3.6 Step Two: Find Active List Neighbors .....	51
3.7 Step Three: Density, Average Velocity, Discomfort .....	53
3.8 Step Four: Find Speed.....	53
3.9 Step Five: Find Unit Cost .....	54
3.10 Step Six: Find Potential .....	55
3.11 Step Seven: Find Final Velocity .....	56
3.12 Step Eight: Move the Units.....	57
4 Comparison .....	59
5 Conclusion: .....	65
5.1 Results.....	65
5.2 Future Works .....	65
5.3 Possible Sources of Error.....	66
6 References.....	67



Here are some examples of CIG-C in action.



Here is 8000 Units finding optimal paths through a dynamic environment in real time.



## **Abstract**

This paper explores CIG-C, a two-dimensional crowd simulation method that uses velocity fields stored inside a quad tree to efficiently move large quantities of crowd members in real-time, interactive framerates. The method is based off of the Continuum Crowds system that attempts to solve the same problem by using a discrete, uniform grid. Using a grid of cells contained within a quad tree, the CIG-C method creates a final velocity field that moves crowd members in a realistic path toward their goal. The CIG-C method is shown to be more stable, accurate, and efficient in many key cases than its Continuum Crowds counterpart. Throughout this paper, the fundamentals of path finding and crowd simulation will be defined and described, leading up to the more modern approaches and methods that directly influenced the development of CIG-C.

# **1 Introduction**

## ***1.1 The Basics***

Path planning is a common problem, encountered in some form or fashion in virtually all real-time simulations. While finding an optimal path is a complicated process even on its own, the problem is exasperated by increasing the number of agents and increasing the size of the search area. Planning an optimal path through terrain with dynamic obstacles (such as other moving agents) requires a call to a path finding algorithm once per actor per frame. Each actor's path affects every other actor's path in sequence, making the order of path finding important. While one actor trying to find a path to its goal is doable, the problem increases in complexity at a geometric rate as more actors try to find paths through the same area. For all but the smallest crowds, corners must be cut to insure real-time framerates.

## ***1.2 Some Uses for Path Planning***

Applications of large-scale path planning are numerous. Real-time strategy games and military simulations cannot exist if the user has no way of commanding his or her troops from location to location. In some RTS games, players may be controlling hundreds, even thousands, of troops, and those troops are expected to be able to walk/swim/fly across the game world in a realistic and timely manner. Crowd simulations used for determining evacuation plans from high-occupancy venues such as stadiums and theaters need to be able to realistically show the tendencies of groups of people as they attempt to

leave the area. In these kinds of simulations, sometimes real-time speeds are not required, instead favoring increased accuracy and complex agent behaviors. Programs that model wildlife require extensive path planning in the form of complex group behaviors. Systems that model herds, schools, and flocks are all examples. Fluid simulations make extensive use of path planning by the use of force and velocity fields. Such fields control the speed and flow of the particles that make up the fluid. Regardless of the use, one trait unifies all applications; as more agents are added to the system, the problem quickly becomes intractable.

## 2 Previous Work



*Figure 1: Boids flocking around with each other, from Reynolds' flock and herd simulation, 1987 [1].*

### ***2.1 Flocks, Herds and Boids***

A good starting point when considering the complex issues of crowd movement is flock and herd behavior, specifically the behavior outlined in [1]. Reynolds' simulation is a combination of a particle system with rule-based actor objects. Most particle systems treat a particle as a point in space with no orientation and very simple individual behavior. A particle is generally capable of changing only its own internal states, such as position, life span, color, etc. By adding an orientation, model, and a set of behavioral rules to each particle, Reynolds changes the particles into entities known as "Boids," a collection of actors that are now capable of interacting with each other as well as with their own internal states [1].

The thing that makes the Boids behave convincingly like a flock of birds is their adherence to a set of rules. These behavioral rules can be specified by the animator, and they are evaluated each frame per Boid in order of decreasing precedence. Such a list of rules is stated below:

Collision Avoidance - Do not run into another Boid (or any other object for that matter).

Velocity Matching - Attempt to match velocity with nearby flock mates.

Flock Centering - Attempt to stay close to nearby flock mates.

Following these rules generates reasonably believable flock behavior. The Boids accrete into groups, which eventually conglomerate into a large flock. The Boids stay close together, making sure not to run into each other or anything else. When a large flock heads into an obstacle, it is possible for the flock to become divided, and later the flock can, and does, merge back together. The system works without an animator having to specify a timed path for each flock member, and it works along with a dynamic environment [1].

Reynolds' method differs from previous methods in that there is no globally-held concept of the flock. Previous methods that simulated flocks used a "follow-the-leader" strategy in which a single actor was designated the leader, or a central force model which specifies a point in space to be the center of the flock. Such previous models were unrealistic in their design; a real flock of birds has no leader, no specific center or target,

and the birds are certainly not cognizant of the specific speeds and orientations of each and every other flock mate. The system that Reynolds created has no need for such global information. Each Boid need only be aware of the few neighboring Boids in order to exhibit correct behavior. The movement displayed by the flock of Boids is called emergent behavior, where complex behaviors are automatically derived from various combinations of the smaller, simpler rules that make up the "thought" process of an actor. This emergent behavior is the very essence of Reynolds' flocking system. In fact, the reduction of agents' perceptions down from global knowledge to limited local intel is necessary to generate the correct behavior [1].

While the behavior of the system is controlled by what rules are fed into the system, the computational complexity and performance is predominantly controlled by the chosen method of collision avoidance. A naive approach results in all Boids comparing themselves against all other Boids every single frame, simply to determine which Boids any given Boid will actually have to interact with at the time. This results in  $O(N^2)$  complexity, which severely limits the potential size of the flock. With the introduction of spatial partitioning (or similar methods of localized collision avoidance) the complexity can drop down to  $O(N)$  [1]. Similar concepts of particle-like actors and spatial partitioning will be used later in CIG-C.

## ***2.2 Dijkstra's Algorithm***

Another important consideration when trying to model crowd movement is, of course, path finding. Here, the method conceived by E. W. Dijkstra in his 1959 paper, "A Note

on "Two Problems in Connexion with Graphs" is examined in detail [2]. Dijkstra's algorithm is capable of finding an optimal path between two nodes on a fully connected discrete graph. This process, if performed naively, incurs a complexity of  $O(N^2)$ , where  $N$  is equal to the number of nodes in the graph; however, with the proper use of the heap data structure, this time can be reduced to  $O(N \log N)$ . Consider the following scenario:

On an eight by eight chess board, the player's rook begins at space A8, and the enemy king stands at space H1. The player's goal is to capture the enemy king, and he/she is permitted only to move the rook. Assuming that the enemy king never moves, what is the minimum number of spaces that the rook must move across in order to capture the king? Furthermore, what is the shortest path from the rook to the king?

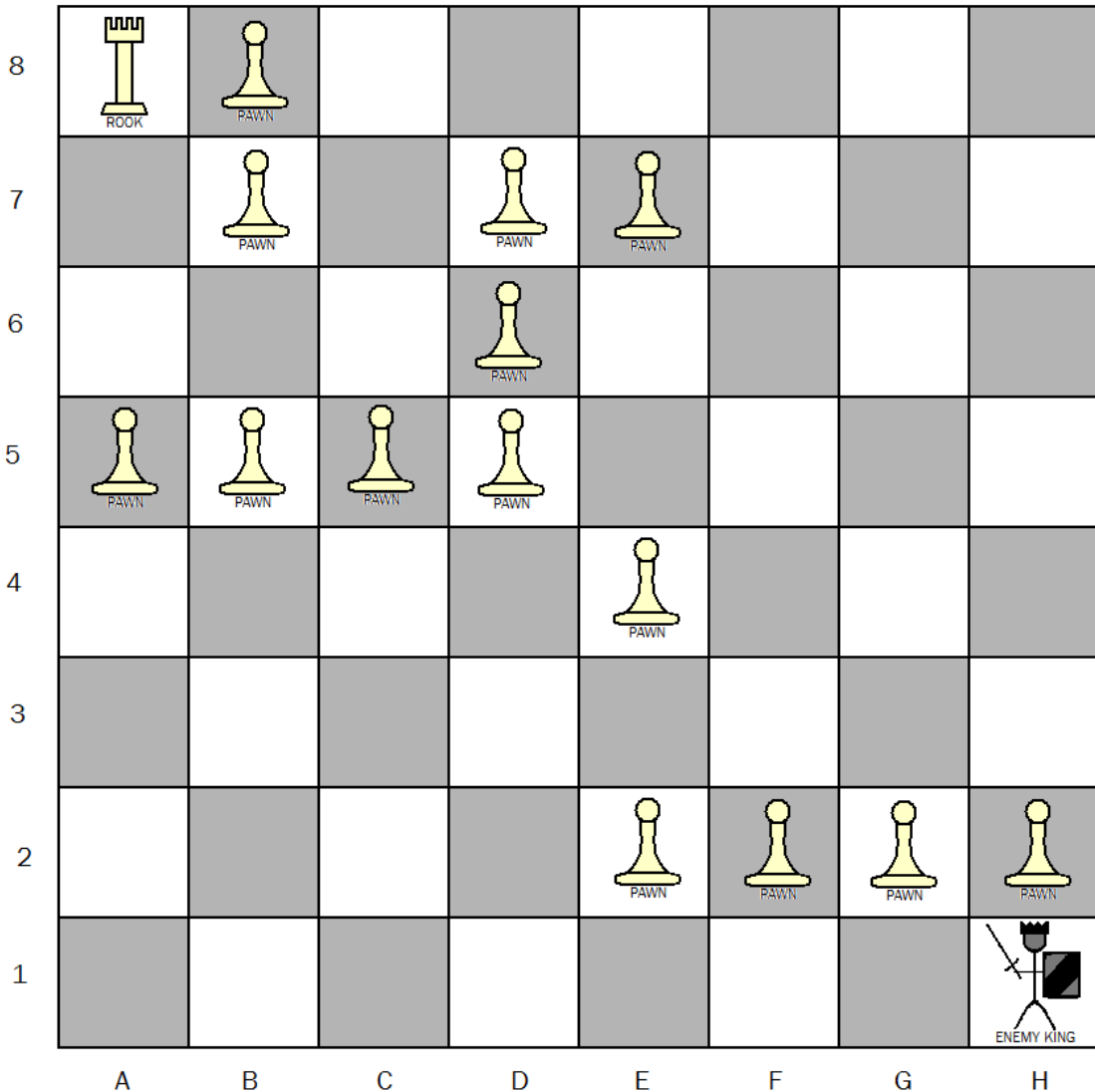


Figure 2: The player may only move the rook in order to take the enemy king. As are the rules of chess, a rook cannot move diagonally, nor can it move through occupied spaces.

In order to solve this problem, Dijkstra's algorithm performs a breadth-first search, starting from the king's position, ending when the rook's position has been explored (or in a more general sense, when all spaces have been explored). The procedure to determine the shortest path and its length is as follows [3]:

- 1) Set the starting position's value to zero, and set the values at all other positions to



- infinity.
- 2) Make two lists: the FOUND list and the WAVEFRONT list. Place the starting space in FOUND. Place all unobstructed spaces that are adjacent to the starting space in WAVEFRONT, their values set to one (unobstructed means empty, diagonal spaces are not adjacent).
  - 3) While WAVEFRONT is not empty:
    - a) Pop the front space off of WAVEFRONT and push it onto FOUND. This space is considered our CURRENT space.
    - b) Place all unobstructed spaces not on the FOUND list that are adjacent to CURRENT into WAVEFRONT, setting their values to one plus CURRENT's value. These insertions should be in order from smallest to largest, with the smallest-valued spaces being at the beginning of the list. If a space is inserted into WAVEFRONT that is already in WAVEFRONT, then use the version with the smallest value and discard the other from the list (duplicates are not permitted).
  - 4) End.

The result of this algorithm is shown below.

















8	 22 ROOK	 PAWN	16	15	14	13	14	15
7	21	 PAWN	17	 PAWN	 PAWN	12	13	14
6	20	19	18	 PAWN	12	11	12	13
5	 PAWN	 PAWN	 PAWN	 PAWN	11	10	11	12
4	10	9	8	7	 PAWN	9	10	11
3	9	8	7	6	7	8	9	10
2	8	7	6	5	 PAWN	 PAWN	 PAWN	 PAWN
1	7	6	5	4	3	2	1	 ENEMY KING
	A	B	C	D	E	F	G	H

Figure 3: The costs on the pawn's spaces are all infinity.

With the values assigned across the chess board, the minimum number of spaces needed to be traveled across in order to take the king now appears as the value located at the rook's position. In this case, the answer is 22. Tracing the path that grants this number is trivial; starting at the rook's position, always move in the direction of an adjacent value that is smaller than the value at the rook's current position. The optimal path should look like this:

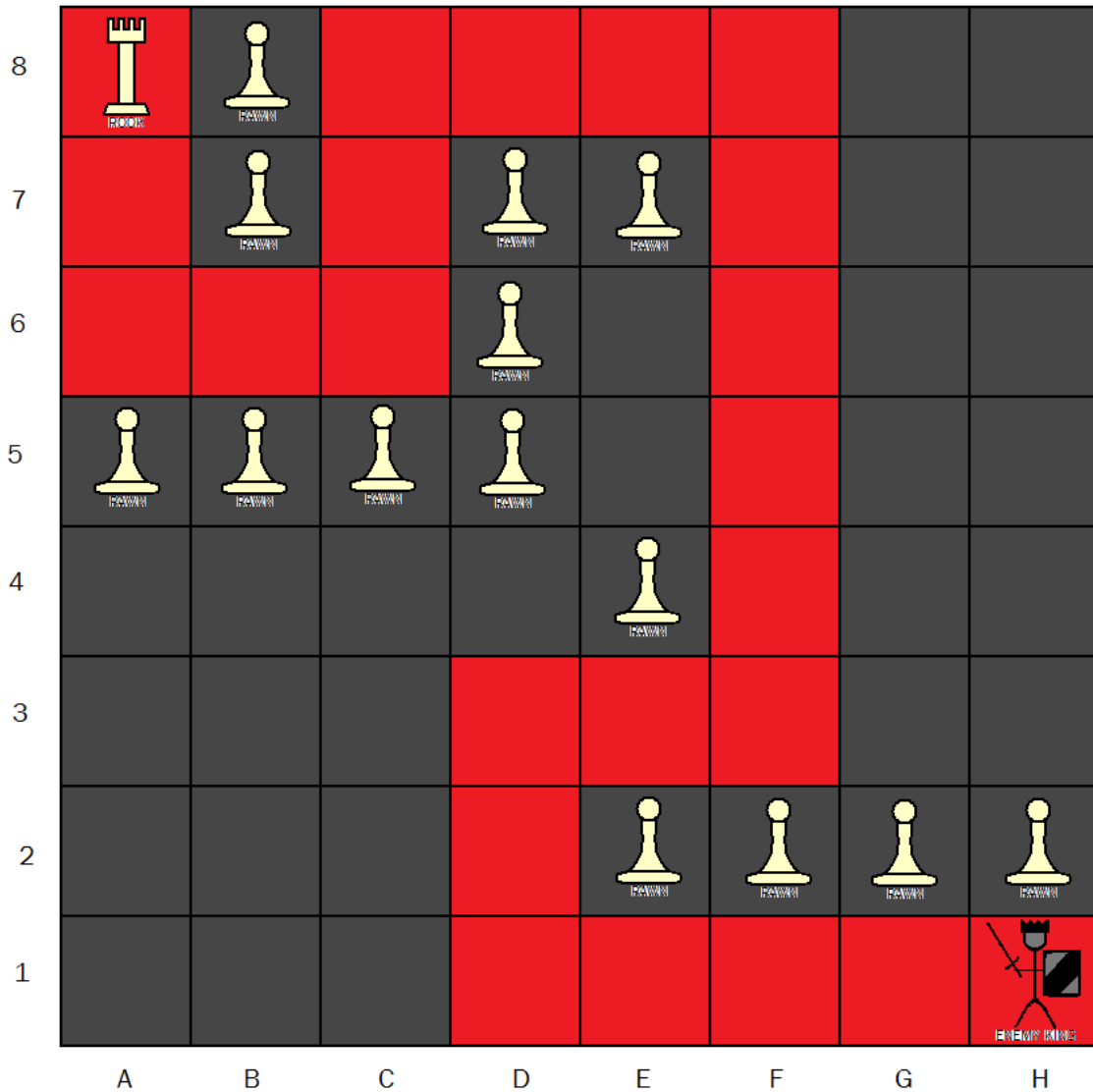


Figure 4: The optimal path from the rook to the king is displayed in red.

In the previous example, the cost of moving from one space to another was uniform; it was always one. Dijkstra's algorithm works just as well on graphs where the cost (distance, time, resistance, etc.) between each node is non-uniform, so long as the cost is known. The algorithm can easily be extended into three-dimensional graphs and higher. Because the above stated steps of the algorithm assumed that WAVEFRONT was a sorted list, the complexity of the algorithm is  $O(N^2)$ , where N is equal to the number of

spaces on the graph. This is because all  $N$  spaces must be visited (worst case), and all visited spaces must be inserted into the WAVEFRONT list in a smallest-to-largest order, a process that itself incurs an  $O(N)$  cost. However, if WAVEFRONT is made to be a heap data structure (the root always being the node containing the smallest value), then the insertion cost drops from  $O(N)$  down to  $O(\log N)$ , lowering the total cost of Dijkstra's algorithm to  $O(N \log N)$  [2]. Though this is a notable improvement, the best way to speed up the algorithm is to lower the number of nodes that must be searched across.

### ***2.3 The A\* Algorithm***

Algorithms such as A\* do exactly this. A\* is a generalized form of Dijkstra's algorithm that uses heuristics to eliminate sections of the graph during search time. With a good heuristic, the complexity of A\* can be lowered to  $O(N)$ , while a bad heuristic causes A\* to perform a breadth-first search of the entire graph, thus precisely mimicking Dijkstra's [4]. That being said, there are uses for traversing the entire graph as Dijkstra's algorithm does. Dijkstra's method for finding an optimal path across a graph generates a ton of extra data in the process of finding the shortest path. In figure 3, all spaces contain a value representing the distance from that space to the king's space. This extra data can be directly used to find additional optimal paths at no added cost. For example, were there another rook at space E5, after calculating the cost for the first rook, it would immediately be known that the second rook's shortest path is 11 spaces to the king. This illustrates the point that one iteration of Dijkstra's algorithm can be used for any number of actors trying to find a path to a single, shared goal, unlike methods such as A\* that must be executed once per actor [5].

## 2.4 Fast March

The chess board example assumes very rigid movement constraints; the rook can only move in the four cardinal directions. What if our simulation calls for a more continuous graph that allows actors to move freely in all directions?

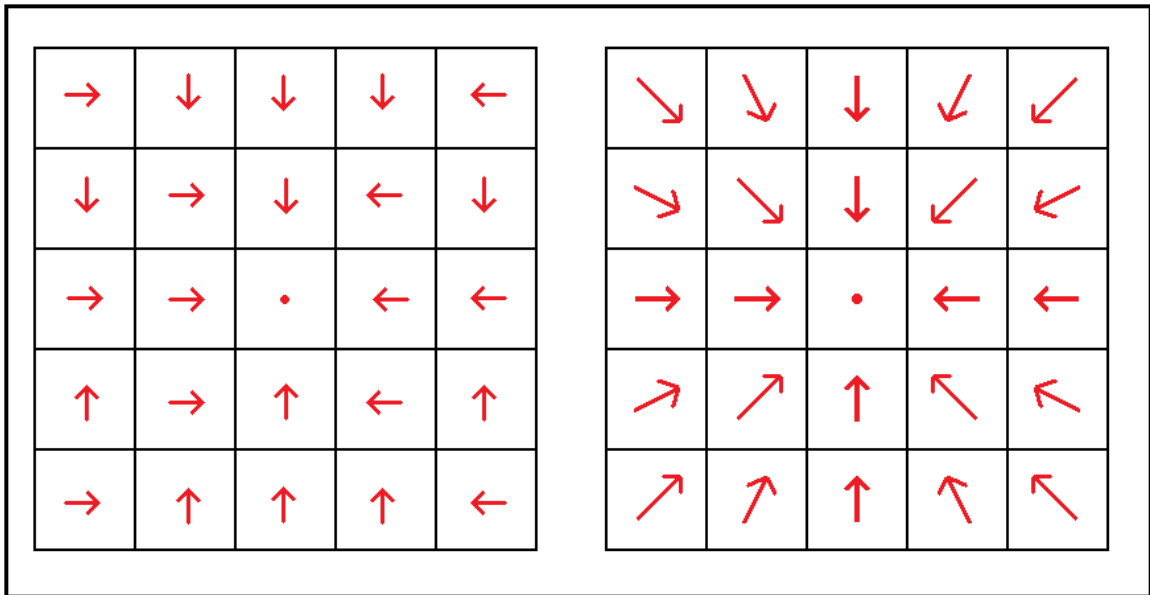


Figure 5: Without any other considerations, a method like Dijkstra's algorithm would generate a velocity field similar to the graph on the left. To generate a more continuous field, like the one on the right, extra steps must be taken.

A first glance might prompt one to believe that a smoothing technique is in order.

Perhaps a bilinear interpolation across the field would do the trick? Unfortunately, no matter what is done to smooth the left graph out, if actors are moving across it toward the center, they will still exhibit signs of Manhattan-style path finding (jagged, right-angle trajectories). In order to solve this problem while still maintaining the Dijkstra-like single pass behavior, we enter the realm of level set theory and delve into a method known as Fast March. The Fast Marching algorithm, as first described in J.A. Sethian's

"A fast marching level set method for monotonically advancing fronts" [6], uses a finite difference approximation to solve the Eikonal equation that describes a potential field in which the solution does not change signs. This, in my opinion, is a horrible way to describe Fast March. A much simpler way to look at things is to consider it a version of Dijkstra's algorithm with an additional intermediate step:

### FAST MARCH

- 1) Set the starting position's value to zero, and set the values at all other positions to infinity.
- 2) Make two lists: the FOUND list and the WAVEFRONT list. Place the starting space in FOUND. Place all unobstructed spaces that are adjacent to the starting space in WAVEFRONT, their values set to C, where C is equal to the movement cost in that space's direction (unobstructed means empty, diagonal spaces are not adjacent).
- 3) While WAVEFRONT is not empty:
  - a) Pop the front space off of WAVEFRONT and push it onto FOUND. This space is considered our CURRENT space.
  - b) Solve the following quadratic equation, finding the largest, positive real root. If none exists, then move onto step iii:
    - i) Find the directions of the less-costly adjacent spaces on the graph along both the x and y axes:

$C_i =$  Traversal Cost in direction  $i$  from CURRENT

$V_i =$  Value at space in direction  $i$  from CURRENT

$dirX = \arg \min_{i \in \{W, E\}} \{V_i + C_i\}$

$dirY = \arg \min_{i \in \{N, S\}} \{V_i + C_i\}$

$i \in \{W, E\}$

$i \in \{N, S\}$

- ii) Use these directions to solve the following quadratic function for its largest real root (solve for  $M$ ). If either  $dirX$  or  $dirY$  is undefined because both neighbors have an infinite cost, then simply drop that dimension out of the following equation:

*Equation 1: Finite Difference Approximation [5, 6, 9]*

$$\left( \frac{M - V_{dirX}}{C_{dirX}^2} \right)^2 + \left( \frac{M - V_{dirY}}{C_{dirY}^2} \right)^2 = 1$$

- iii) If the largest real  $M$  is found, the value of CURRENT is set to that  $M$ .

- c) Place all unobstructed spaces not on the FOUND list that are adjacent to CURRENT into WAVEFRONT, setting their values to C plus CURRENT's value, where C is equal to the movement cost in that space's direction. These insertions should be in order from smallest to largest, with the smallest-valued spaces being at the beginning of the list. If a space is inserted into WAVEFRONT that is already in WAVEFRONT, then use the version with the smallest value and discard the other from the list (duplicates are not permitted).
- 4) End.

Do not be fooled into thinking that there is nothing else to Fast March; the pseudocode written above is a simplification geared toward a specific use rather than a full description of the inner mysteries of Fast March. The proof of the validity of the Fast Marching method goes beyond the scope of this work, and thus the reader is referred to [6] for a full description of the mathematics behind Fast March. The accuracy of Fast March is determined by the discretization of the graph. The coarser the graph is, the less

accurate the method will be [7]. Fast March has many applications, from fluid mechanics to computer graphics. Some of the most recent uses of Fast March can be seen in crowd simulations.

## ***2.5 Continuum Crowds***

### ***2.5.1 Brief Overview***

The previous work most influential to CIG-C is the "Continuum Crowds" paper [8]. In the Continuum Crowds system, based on continuum dynamics, a dynamic potential field integrates global navigation with moving obstacles such as other crowd members, quickly and effectively solving for the movement of large crowds without excessive use of explicit collision avoidance. The notion of agent-based path planning is disposed of, in favor of treating agents as mindless particles that an underlying field acts upon. These particle-agents shall henceforth be known as Units. This field is a uniform grid made up of square cells. In each cell, data such as density, flow speed, and velocity is recorded each frame, which is used to find a movement cost from a cell to each of its four neighbors [8].

Using the Fast Marching algorithm, a dynamic potential field is created and recorded inside the cells of the grid. Final velocity values are assigned to each cell by tracing the gradient of the potential field in the upwind direction, ultimately using the earlier-calculated speed values found in the corresponding direction. Units are moved based on the final velocity value found in the cell they currently occupy. To correct for error, the



last step of the algorithm is to enforce the minimum distance between Units, insuring that they do not overlap with each other [8].

In the Continuum Crowds system, a Group is defined as a set of Units that share a common goal. As such, for each Group, an additional Unit Cost field and dynamic Potential field is required.

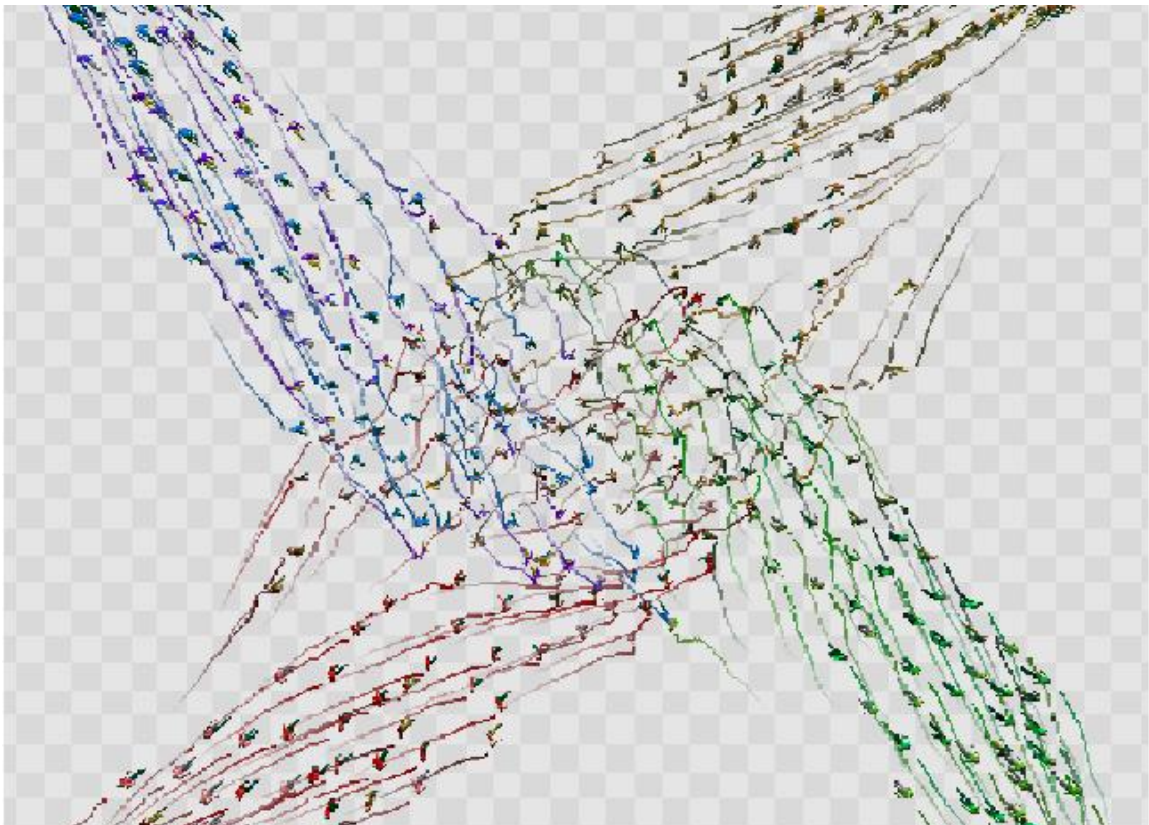


Figure 6: Four groups of Units in a Continuum Crowds implementation cross paths. Image courtesy of [8].

### ***2.5.2 The Governing Hypotheses***

Four hypotheses create the guidelines on which Continuum Crowds is based [8]. The first hypothesis states that each crowd member is trying to reach a geographical goal. This hypothesis seems obvious, but many common crowd situations are incapable of

specifying a single goal. Take for example a group of people browsing in a store; they have no specific goal or driving function, simply the will to look around. As such, a "browsing" simulator is not an application for Continuum Crowds.

The second hypothesis states that crowd members will move at the maximum possible speed appropriate to their current situation. Placing it in the context of people in a city, this hypothesis becomes intuitive. A person has a maximum, unimpeded speed, which is their walking speed on flat terrain. Note that even though it is recognized that people can sprint faster than they can walk, the walking speed is considered the maximum speed here because most people in a city prefer to walk than to sprint, and as such they spend the vast majority of their travel time walking. If a person is in a dense crowd, then that person can only walk as fast as the surrounding people are walking. If a person is walking up a steep hill, then their walking speed will be appropriately slowed. Thus, a person's speed is controlled by a linear interpolation of crowd flow speed and terrain speed.

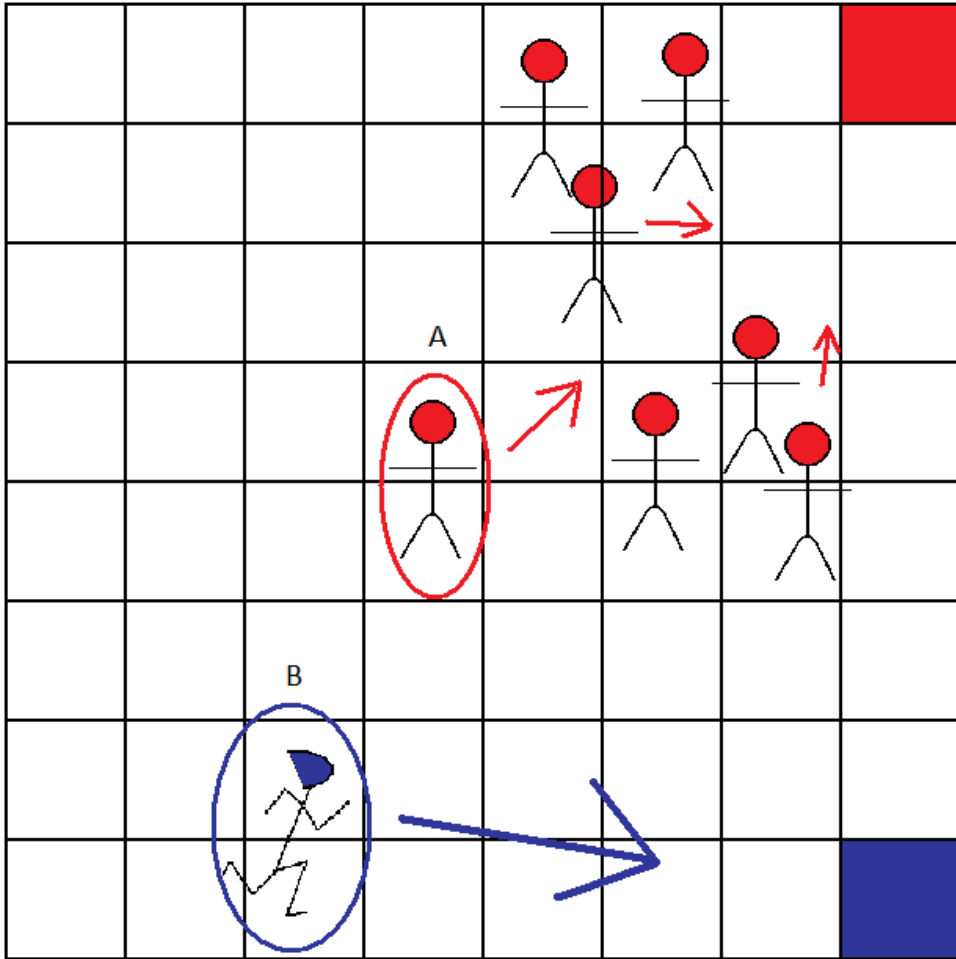


Figure 7: Unit A is behind a crowd of bunched up Units, making his speed slower. Unit B's path is unobstructed, enabling him to move at his maximum speed.

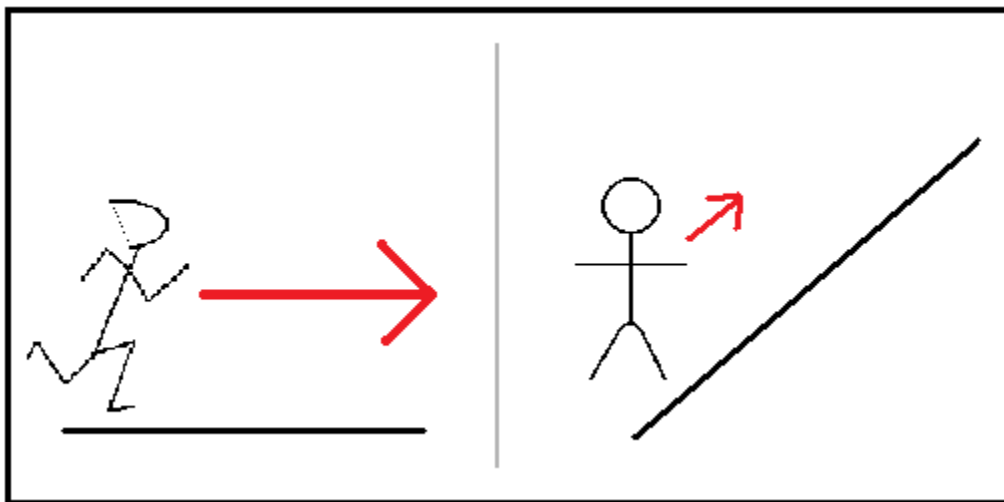


Figure 8: A Unit moves more quickly across flat terrain and slower across steep terrain.

The third hypothesis introduces the concept of the discomfort field. An area with high discomfort is less preferable to a crowd member than is an area with low discomfort. Take, for instance, a meercat and a charging rhinoceros on a large grassy field. Rather than waiting for the last second to leap to safety, the meercat will undoubtedly move out of the rhinoceros's deadly path well before it gets trampled. Such a scenario can be modeled with a discomfort field. Simply project the rhino's density in the direction of its velocity onto the discomfort field. Now, the meercat will avoid the path of the rhino just as sure as it will avoid the rhino itself.

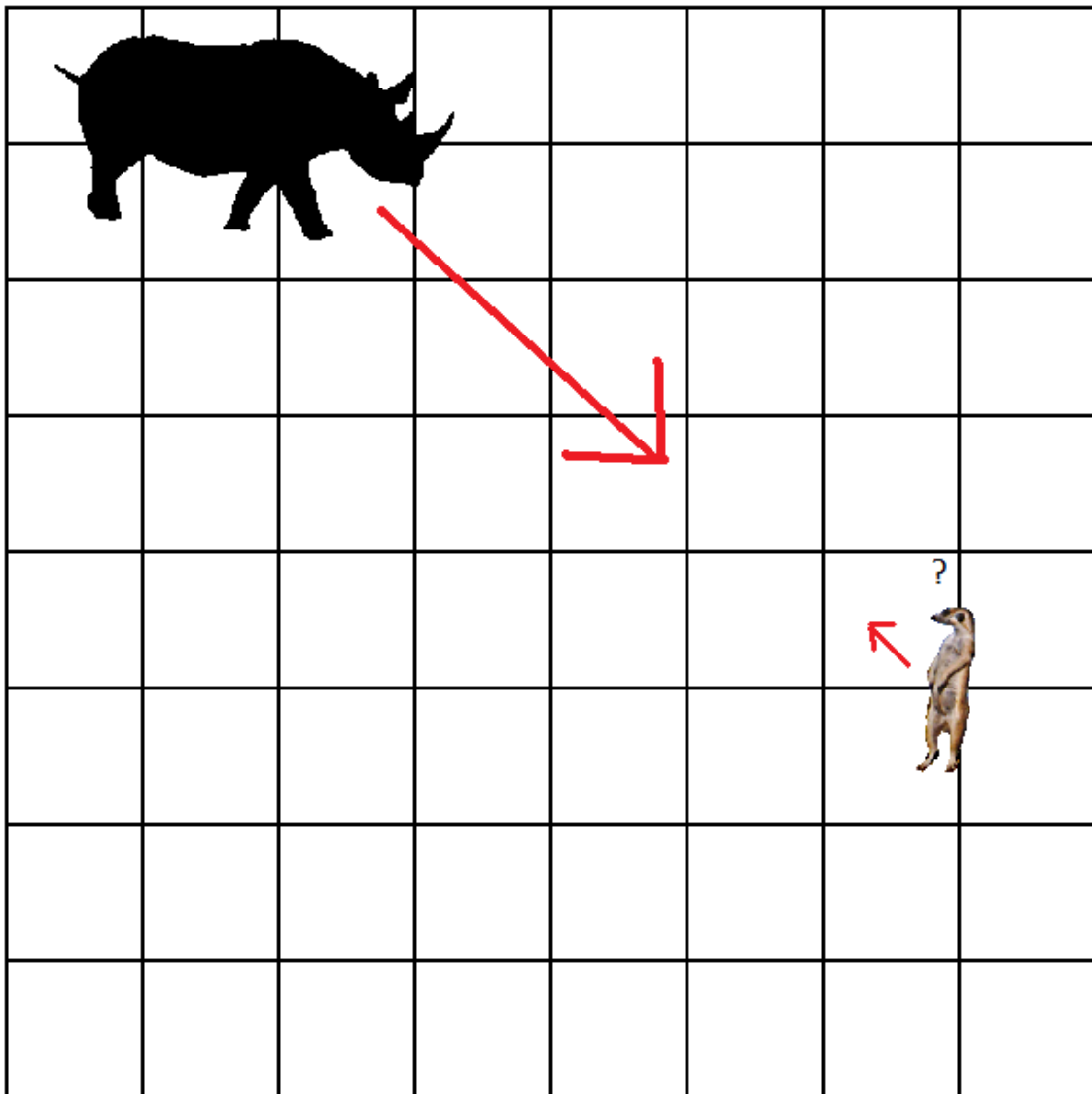


Figure 9: The hapless meercat does not avoid the rhino because its density is currently not blocking its path.

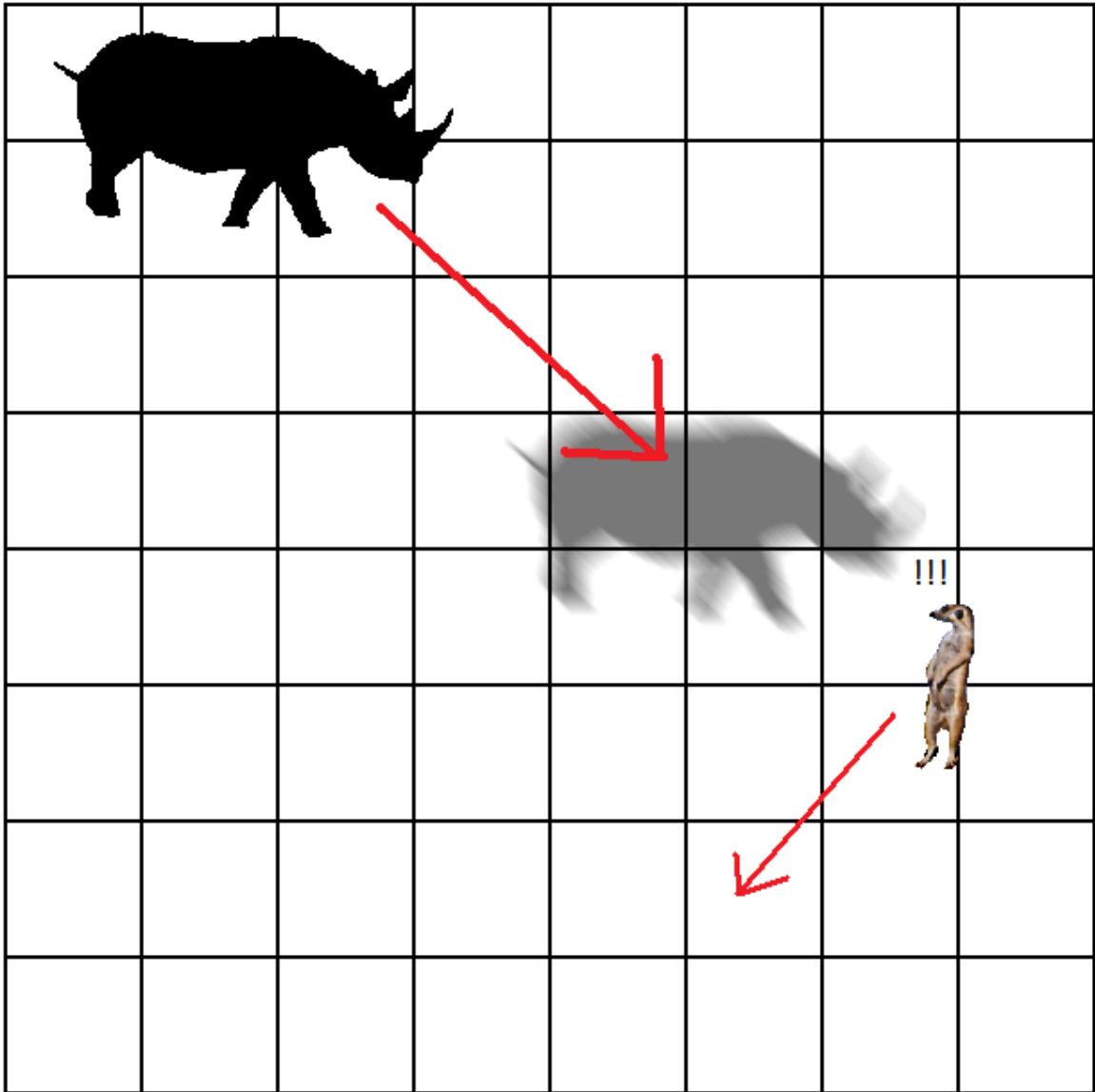


Figure 10: With the addition of the discomfort field, the meercat now sees the impending danger and promptly changes course.

The fourth hypothesis states that when a crowd member is trying to find a path to its goal, it will pick the path that minimizes a linear combination of the following three factors: the length of the path, the amount of time to the destination, and the discomfort field, per unit time, along the path. Basically, a crowd member will pick the fastest, shortest, and

most comfortable path it can find in order to get to its goal. To make the equation fit specific scenarios more appropriately, three user-defined constants scale the strength of each one of the three contributing factors. The constant  $\alpha$  affects the path length,  $\beta$  affects the time across the path, and  $\gamma$  affects the discomfort felt across the path.

$\int_P C$  , where

$$C \equiv \frac{\alpha f + \beta + \gamma g}{f}$$

Equation 2:  $P$  represents the current path. The integral is taken with respect to the path's length. Equation taken from [8].

### 2.5.3 Basic Walkthrough

A basic walkthrough of the simulation is as follows [8]:

For each frame:

- Convert the Units to a density field.

For each Group:

- Construct the Unit Cost field.

- Construct the dynamic Potential field, and find its gradient.

- Update the Unit's locations.

- Enforce the minimum distance between Units.

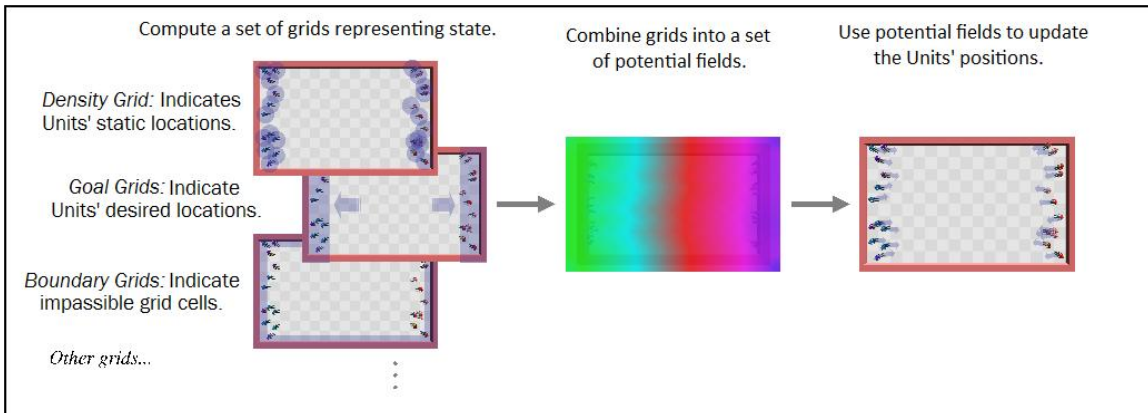


Figure 11: General algorithm overview. Image borrowed from [8].

### 2.5.4 Results

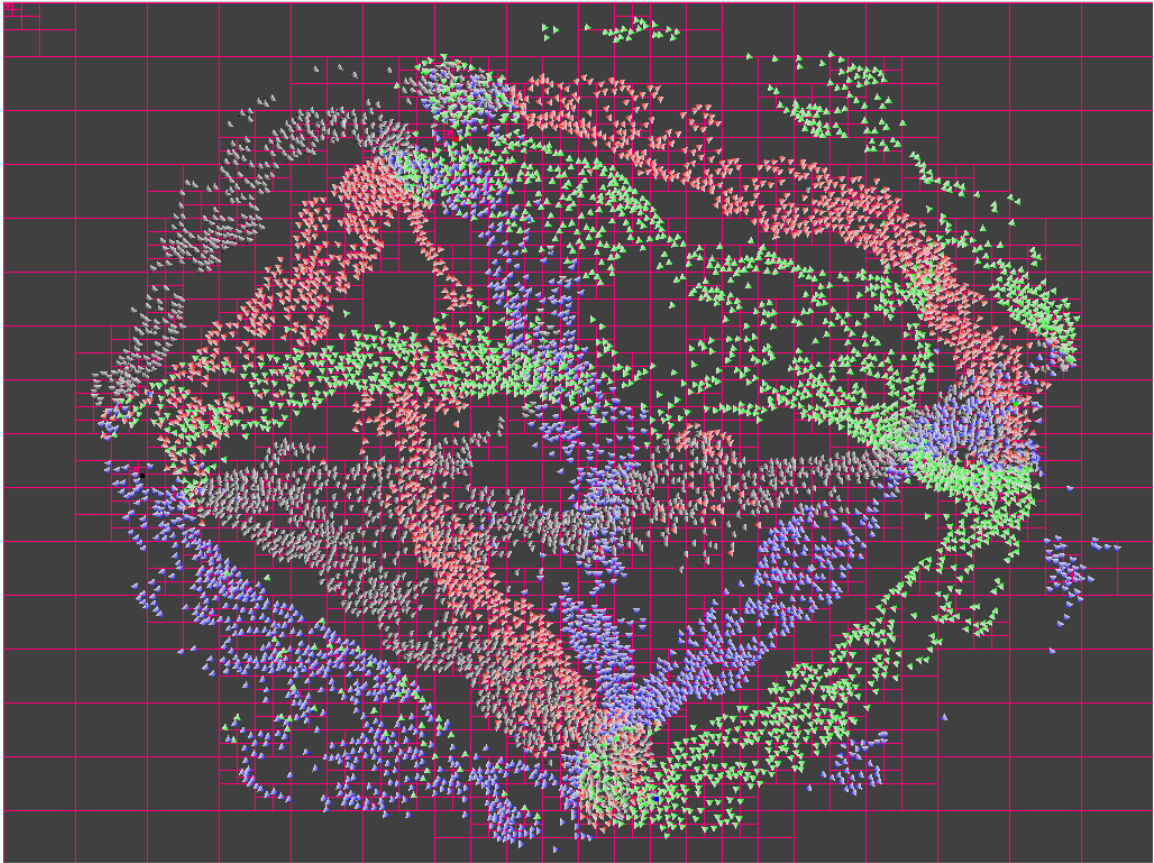
The Continuum Crowds approach yields very impressive results, generating large, interactive crowds (10000+ Units) at real-time framerates. However, there is room for improvement here. In the Continuum Crowds simulation, the size of the cells must be uniform. Using large cells has the advantage of increased speed and lighter calculations, but this is at the cost of accuracy, and can result in "dumb" and rigidly-moving crowds. Indeed, if the cell size is too large, the minimum distance reinforcement step begins to become a quadratic collision-detection problem rather than a nice linear one, due to the fact that there can be way too many Units in a single large cell, which can greatly slow down the system.

On the other side of the coin, if the cell size is too small, then there will be too many cells, and the simulation will simply run too slowly. Thus, a user must estimate in any given implementation how crowded he or she expects the most congested areas will be, and the cell size must be a good balance between large and small to account for correct accuracy at high traffic areas and good performance overall. Oftentimes much of the grid

is unoccupied and unused. Many scenarios result in only a few, select areas of heavy Unit concentration, while the remainder of the grid is sparsely populated. Some may wonder if there is a way to get the best of both worlds, high accuracy at key areas coupled with good performance due to the use of fewer cells.



## 3 Our Approach: CIG-C



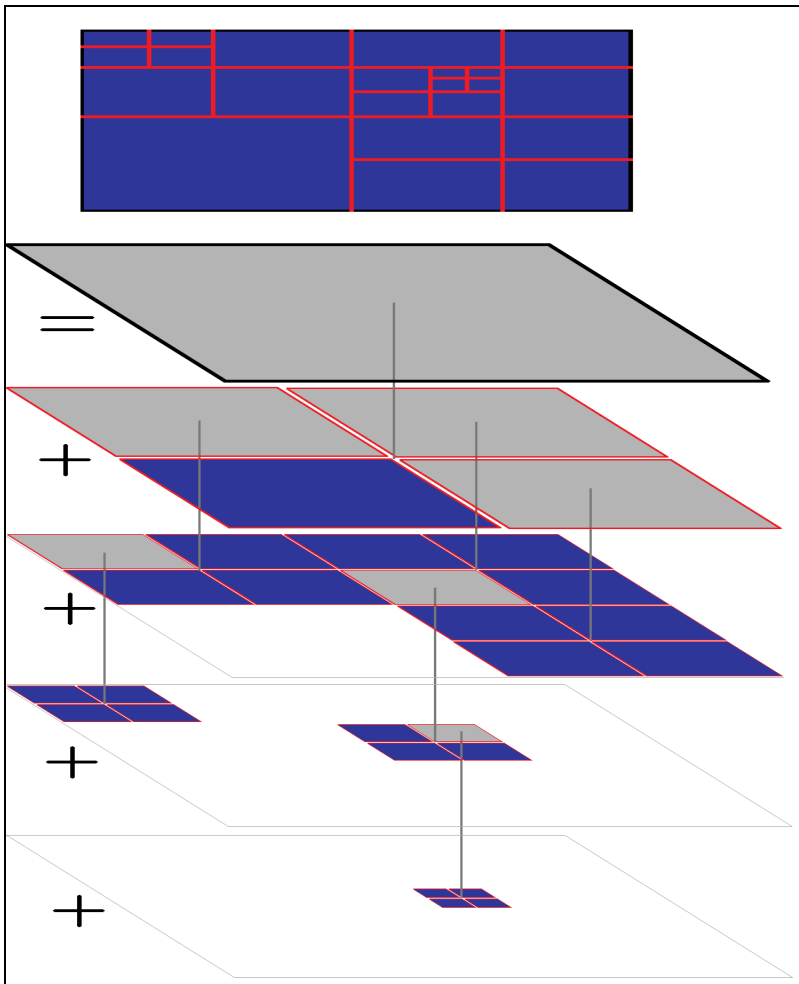
*Figure 12: A CIG-C implementation running with the quad tree lines showing. Four groups, 8000 Units total, to a tree depth of 10 divisions.*

### ***3.1 High Concept***

In case the reader is curious, “CIG-C” stands for “Continuum-Improved-Graph-Crowds.”

The CIG-C crowd simulation method takes an approach similar to Continuum Crowds, yet does away with the notion that all cells are uniformly sized. Instead, the simulation's area is encompassed by a quad tree of cells, going down to a user-defined depth. In the quad tree, cells near the root are larger, with cells at the leafs of the tree the smallest and

the root itself the largest. All cells have the same aspect ratio, which is equal to the aspect ratio of the entire rectangular simulation area as a whole. Rather than iterating through all cells like in Continuum Crowds, CIG-C creates a small subset of cells from out of the quad tree, designated the "Active List". The Active List's cells comprise an area that completely spans the entire simulation space, with larger cells located at areas of low occupancy and smaller cells at areas of high occupancy.

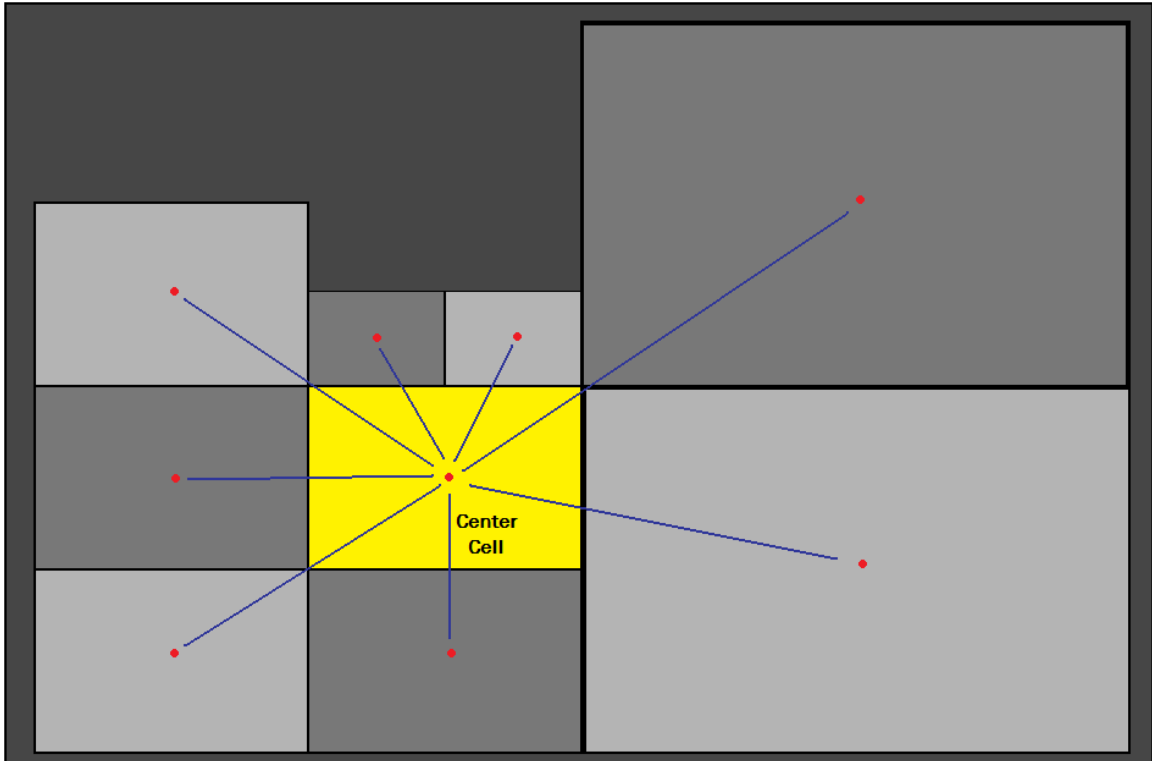


*Figure 13: Blue cells represent active cells. As seen by this graphical summation, the combined area of the Active List completely covers the simulation area.*

### ***3.2 Significant Differences***

Unlike in Continuum Crowds, in CIG-C, while updating the field and thus moving the crowd, it is not necessary to update all cells within the grid, only the ones in the Active List. Because CIG-C only updates a relatively small number of cells each frame, the most expensive part of the simulation, the creation of the dynamic potential field, is minimized in complexity and thus greatly alleviated. This is the key factor in the performance increase that CIG-C offers over Continuum Crowds.

Of course, what is gained in one area must be paid for in another area. In the case of CIG-C, the structure of a cell, and thus its connections with other cells are significantly more complicated than in the Continuum Crowds method. A cell from Continuum Crowds has exactly four neighbors (with the exception of edge cases, which have fewer than four), and because all the cells are of uniform size, the directions and distances from a cell to any of its neighbors are static and deterministic. The directions are always north, south, east, and west, and the distance in all cases is simply the length of the cell. In CIG-C, a cell can potentially have hundreds of neighbors depending on its size and position; furthermore, each cell's number of neighbors can be different. Even cells of the same size can have drastically different neighbor counts. On top of that, the directions between a cell and its neighbors are no longer set to the four cardinal directions, nor are they a constant, pre-calculated length.



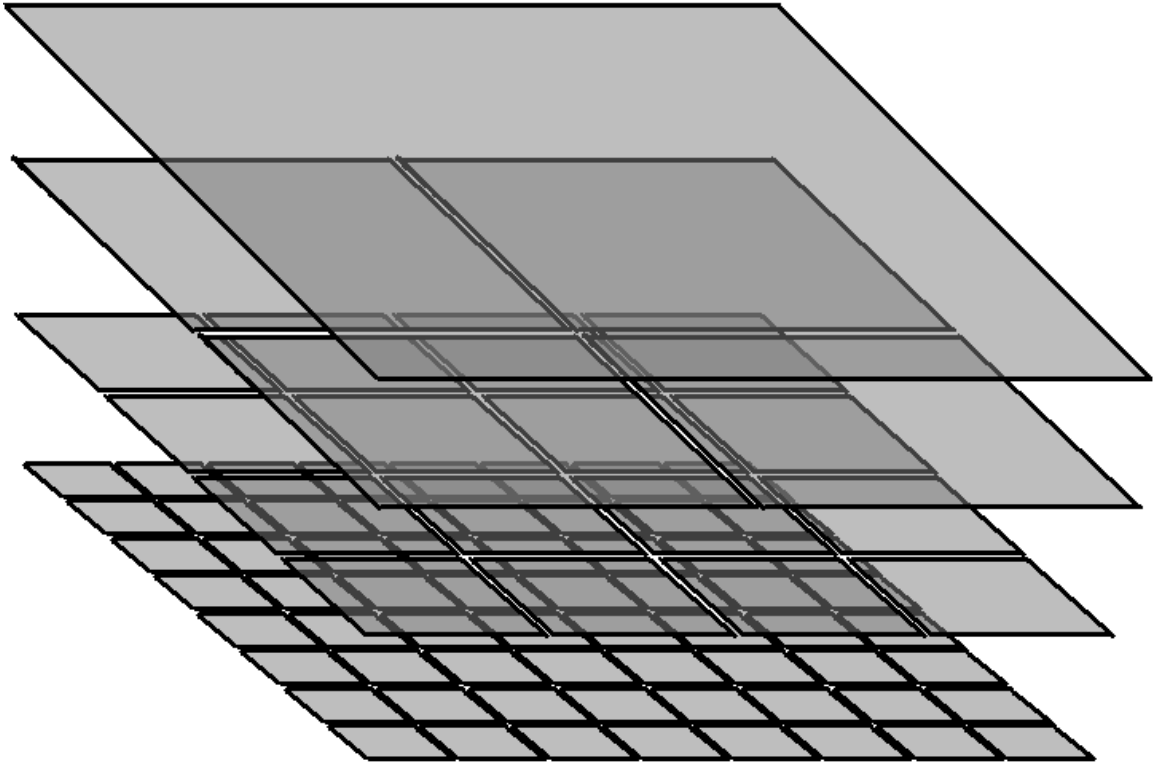
*Figure 14: The center cell is connected to eight neighboring cells. The blue lines represent the anisotropic data stored along with the cell pointer leading from the center cell to the adjacent neighbors. Please note that in CIG-C, diagonal cells are considered to be adjacent in this context.*

These directions and lengths must be calculated and recorded each frame for every new Active List. Fortunately, this process is not as bad as it sounds. The additional time it takes to perform these calculations is more than made up for by the savings gained during the dynamic potential field step. Also, even though a vast number of neighbors are possible, most of the time a cell will end up having close to eight neighbors. Due to the increased and varying number of neighbors, the Fast March technique has been replaced by a version of Dijkstra's algorithm. The actual structure of CIG-C is now explored in further detail, starting with the field's creation, followed by a step-by-step breakdown of what happens during a single frame.

### ***3.3 Creation of the Field - Before the Action***

#### ***3.3.1 Tree Structure***

To create the field, first set the dimensions of the field, width and height (this should correspond to the width and height of whatever area the user wants the crowd to be in). This aspect ratio will be maintained across all the field's cells. Next, allocate the space for the cells themselves. The cells will need to represent a quad tree that spans the simulation area. It will have a user-specified depth. This allocation is done to avoid having to dynamically allocate and deallocate memory during execution. After having made all the cells in the tree, go through each level of the tree, linking neighboring cells with cell pointers. At this point, only cells on the same level of the tree can be considered neighbors of each other; these neighbors are specially-designated "same-level" neighbors, and are only used when constructing the list of active neighbors during execution. This completes the creation of our field, and it is only done once at the beginning of execution.



*Figure 15: A graphical representation of a four-leveled quad tree. This structure is allocated once at the beginning of the simulation.*

### ***3.3.2 Breakdown of a Single Frame***

- 1) Frame Start
- 2) Find Active List Neighbors
- 3) Density, Average Velocity, and Discomfort
- 4) Speed Calculation
- 5) Unit Cost Values
- 6) Potential (via Dijkstra's)
- 7) Final Velocity
- 8) Move Units

### ***3.3.3 The Reasoning Behind the Quad Tree***

The whole point of replacing the uniform grid with something else is to take advantage of the special properties of spatial partitioning, namely the highly structured storage of sparse information without wasting memory on empty space. Structures like KD trees accomplish this goal, but they are not as predictable as quad trees in one key sense: their cells' connections to their neighbors. Whereas it is quite a challenge to find connectivity data between cells in a KD tree, it is comparatively trivial in a quad tree structure. Perhaps most important is the fact that in a quad tree, it is a guarantee that any two neighboring cells have a straight-line path from one center to the other that does not cross over any third cell. The same cannot be said about cells in a KD tree.

Proof of Claim:

#### **Situation:**

There exists a 2D rectangular region (called a "Gridspace") with coplanar neighbor Gridspaces of the same aspect ratio. Gridspaces are aligned as such that they are members of the same quad tree.

#### **Hypothesis:**

For any Gridspace  $G$  with width  $W_g$  and height  $H_g$  and any Gridspace  $N$  with width  $W_n$  and height  $H_n$ , there exists a line segment  $L$  such that:

- 1)  $L$  starts at the center point of  $G$  and ends at the center point of  $N$ .
- 2) All points on line segment  $L$  lie inside either  $G$  or  $N$

(In other words,  $L \in G \cup N$  ).

, so long as:

1)  $G$  and  $N$  are neighbors, i.e. the perimeter of  $G$  and the perimeter of  $N$  share more than one point, and

2)  $G$  and  $N$  have the same aspect ratio

(  $W_g * H_g = C * W_n * H_n$  , where  $C \in \mathbb{R} > 0$  ).

**Proof:**

To begin, we will consider the most "extreme" case of  $G$  and  $N$  having met the requirements for adjacency.  $G$  will be located in the farthest corner of  $N$ .

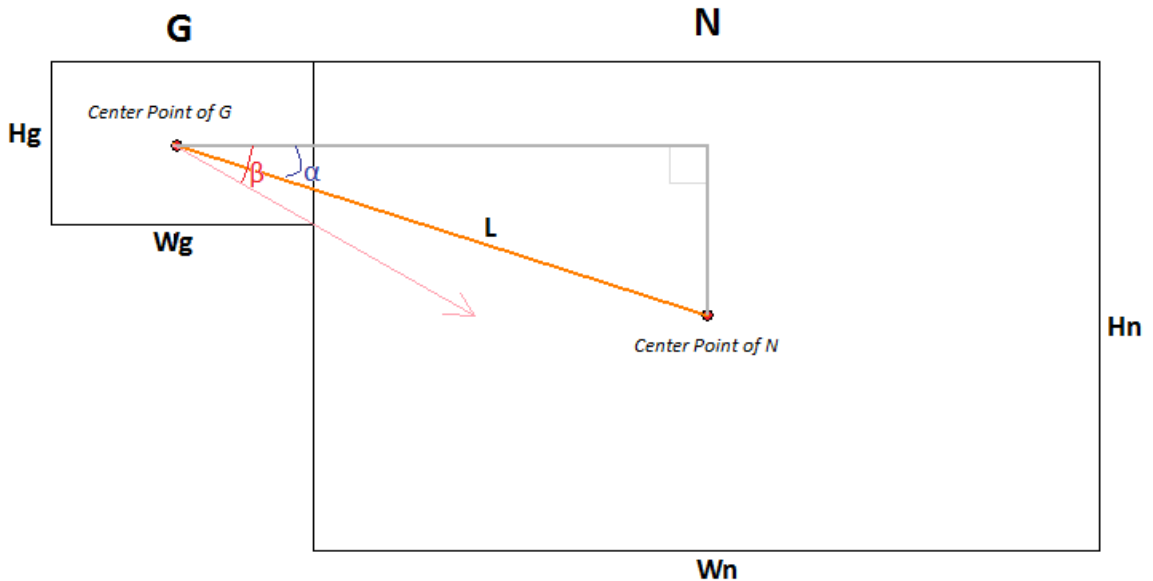


Figure 16: Gridspace  $G$  is at the place that maximizes the angle  $\alpha$  with respect to Gridspace  $N$ . When proven at this position, all other cases will be trivial.



Note: It is assumed that all distances and positions are positive, and degrees are measured in radians.

Before we can proceed, we must describe the variables. Already described above, there is  $W_g$  and  $H_g$  (width and height of Gridspace  $G$ , respectively), and there is  $W_n$  and  $H_n$  (width and height of Gridspace  $N$ , respectively). Line segment  $L$  extends from the center point of  $G$  to the center point of  $N$ . There also exists a variable  $R$  such that:  $R = \frac{W_g}{H_g}$  and

$R = \frac{W_n}{H_n}$ . The variable  $R$  is the aspect ratio of the Gridspaces and is shared by both  $G$

and  $N$ . Another way to express the aspect ratio is through the scalar  $C$ , as described in the hypothesis. Finally, we get to  $\alpha$  and  $\beta$ . The angle  $\alpha$  represents how much the line segment  $L$  must deviate from a "straight shot" to get to the neighbor's center point. It can be expressed as the following equation:

$$\alpha = a \cos \left( \frac{\frac{W_g + W_n}{2}}{\|L\|} \right)$$

Angle  $\beta$  represents the maximum amount of "deviating" the line segment  $L$  is allowed to do before it no longer satisfies the condition that all of its points lie within the union of the areas of Gridspaces  $G$  and  $N$ . It can be expressed as the following equation:

$$\beta = \frac{\pi}{2} - a \tan \left( \frac{W_g}{H_g} \right)$$

, or alternatively:

$$\beta = \frac{\pi}{2} - a \tan \left( \frac{W_g}{H_g} \right)$$

If  $\alpha > \beta$ , then the line segment  $L$  would not lie entirely inside the union of  $G$  and  $N$  and therefore would disprove my hypothesis. However, if the converse were true, that  $\alpha$  is always less than or equal to  $\beta$ , then by definition my hypothesis would be proven correct. Therefore I will show that for any  $C$  one to infinity,  $\alpha$  can never be greater than  $\beta$ .

We start with the equation of  $\alpha$ ,

$$\alpha = a \cos \left( \frac{\frac{Wg + Wn}{2}}{\|L\|} \right)$$

and we perform simple substitutions using the following equalities:

*Equation 3: Simple algebra derives these equations, using the equation  $Wg * Hg = C ( Wn * Hn )$  as a starting point.*

$$R = \frac{W_i}{H_i}, \text{ where } i \in \{G, N\}$$

$$H_i = \frac{W_i}{R}, \text{ where } i \in \{G, N\}$$

$$W_i = RH_i, \text{ where } i \in \{G, N\}$$

$$Hn = \frac{Wg \sqrt{\frac{1}{C}}}{R} \qquad Wn = Wg \sqrt{\frac{1}{C}}$$

$$\|L\| = \sqrt{\left(\frac{1}{2}Hn - \frac{1}{2}Hg\right)^2 + \left(\frac{1}{2}Wg - \frac{1}{2}Wn\right)^2}$$

, reducing the equation to

$$\alpha = a \cos \left( \frac{R \left( \sqrt{\frac{1}{C}} + 1 \right)}{\left( \left( \sqrt{\frac{1}{C}} + 1 \right)^2 R^2 + \left( \sqrt{\frac{1}{C}} - 1 \right)^2 \right)} \right)$$

If we take the limit of  $\alpha$  as  $C$  approaches infinity, it is shown that  $\alpha$  converges to the upper bound:

$$a \cos \left( \frac{R}{\sqrt{R^2 + 1}} \right)$$

Finally it is shown

$$\beta = \frac{\pi}{2} - a \tan \left( \frac{R}{\sqrt{R^2 + 1}} \right) \Rightarrow a \cos \left( \frac{R}{\sqrt{R^2 + 1}} \right)$$

, and since the upper bound is *equal* to  $\beta$ , it is *impossible* that it will ever be greater than  $\beta$ . As shown, the actual length, width, and aspect ratio of each Gridspace is irrelevant, so long as the aspect ratio maintains a constant, positive, real number.

In a similar manner, it can be shown that for any position such that G and N are adjacent,  $\alpha$  is less than or equal to  $\beta$ . Therefore my hypothesis has been proven true.

Note that the above proof does not attempt to prove the four “perfect diagonal” cases, being northeast, northwest, southeast, and southwest connectivity. Those four cases require no proof, as they are trivial; the line drawn from the first Gridspace to the second crosses precisely through the single point connecting the two Gridspaces. This is a fact for the exact same reason as the proof is possible: all Gridspaces have the same aspect ratio.

### ***3.4 Algorithm Overview:***

A frame begins by resetting the field and constructing the Active List. Next, all the cells in the Active List find their respective Active List neighbors. Once the Active List is done finding its adjacency information, each cell's density, average velocity, and discomfort is calculated. Then, a linear interpolation is used to find the speed values inside each cell in the Active List. Using the speeds, distances, and discomforts in any given cell in the Active List, the Unit Cost values are calculated.

The next step is the most costly: finding the potential values. In order to find the potential values at each cell in the Active List, a version of Dijkstra's algorithm is executed, using Unit Cost as the path weights. After the potential values have been found at each cell throughout the Active List, they are used to find the final velocity term in each cell. Finally, using the final velocity vectors appropriate to each Unit, the Units are moved.

### ***3.5 Step One: Frame Start***

At the beginning of each frame, the Active List must be reset and reconstructed. In order to do this, first the field must be wiped clean. Next, the Units are "splattered" onto the field. Finally, in a three-step process, the Active List is constructed from the cells inside the field's tree.

### ***3.5.1 Resetting the Field:***

To reset the field, iterate through the tree, resetting each cell's dynamic values. Each cell must be removed from the Active List. Also, each cell's number of occupants, density, and discomfort values must all be set to zero. Then, the "potential" values must be set to infinity (99999.9f will do in C++). Finally, we must clear the Active List.

### ***3.5.2 Splattering the Units:***

To splatter the Units onto the field, iterate through the Unit list. Look at each Unit's position, and pair it up with the corresponding cells. A Unit is guaranteed to be on more than one cell; in fact, they are guaranteed to be on exactly H cells, where H is the height of the quad tree. For each Unit, keep track of these H overlapping cells. Later, when the Active List has been constructed, exactly one of the H cells will be Active, and that cell will be the proper "current" cell for the Unit.

### ***3.5.3 Pre-Active-List Construction:***

The "pre-Active-List" step prepares the tree for the construction of the Active List. During this step, each cell designated as a goal has its occupant count artificially inflated by a large constant (say, 99999). Going up the tree from each goal toward the root, each subsequent parent cell's occupant count is identically inflated. This is done to force the grid around each goal cell to be divided into the smallest possible discretization. In a similar manner, all cells at a specified level have their occupant counts inflated. This level is designated as the "minimum level" for the tree, and it is a user specified constant.

The minimum level represents the coarsest level of any grid cell that will be allowed to appear on the Active List.

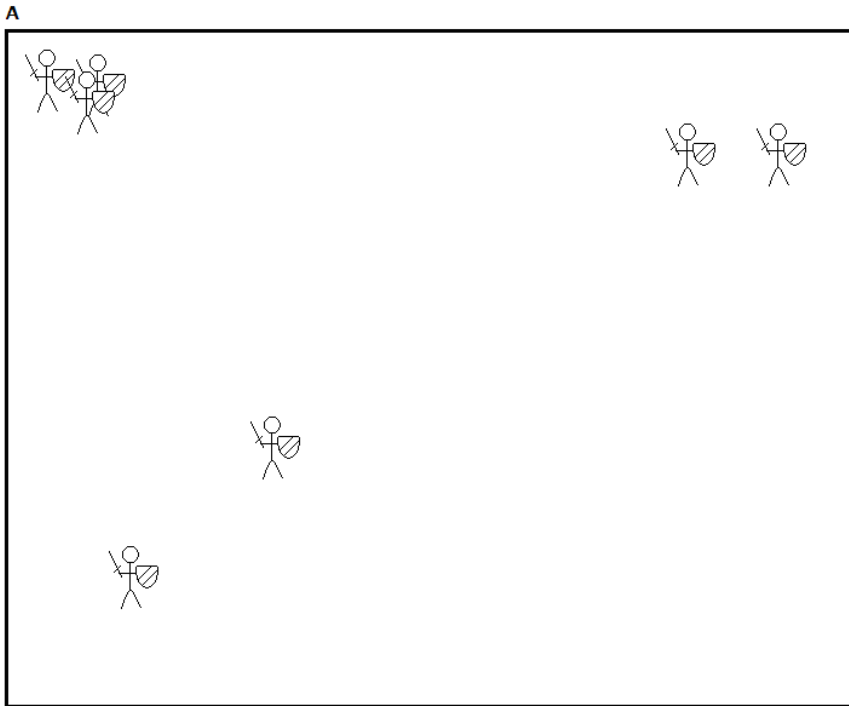


Figure 17: Blue cells represent cells that do not need to be broken down any further, and white cells are cells that have too many Units in them and so must be split into smaller cells. Here, we start in step A, with a top level cell that is filled past capacity.

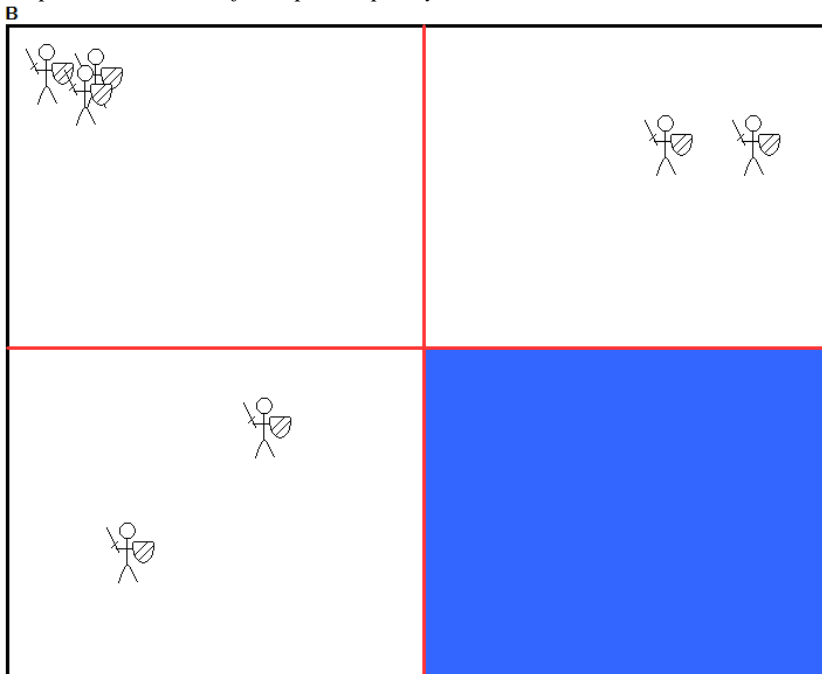


Figure 18: Step B divides the cell into four smaller cells, which finishes the division process for the bottom-right quadrant. The rest of the grid still needs some work.

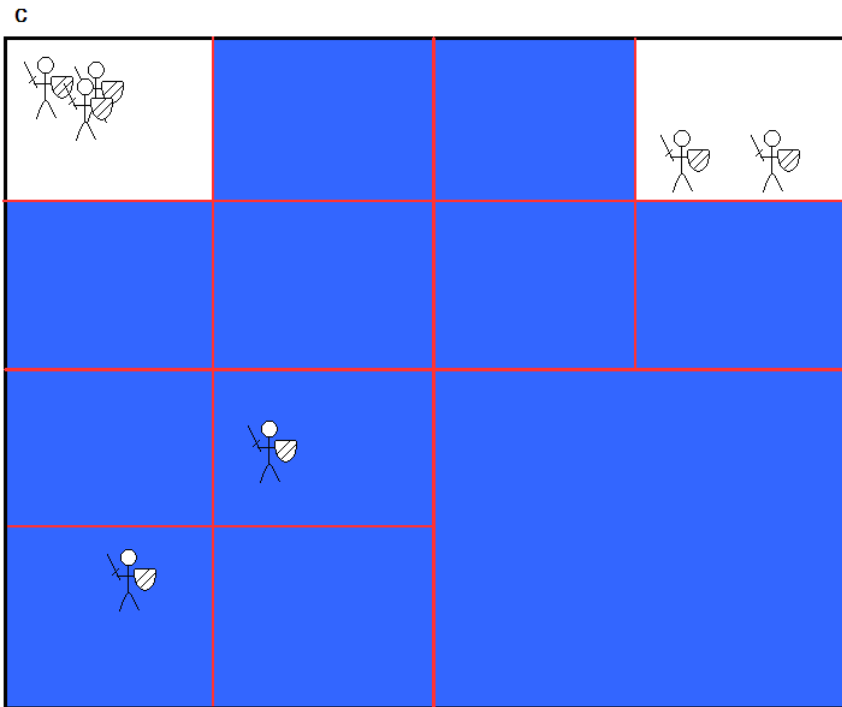


Figure 19: Step C splits the white cells again, which finishes off all the divisions except the top-right-most and top-left-most cells, which are still beyond capacity.

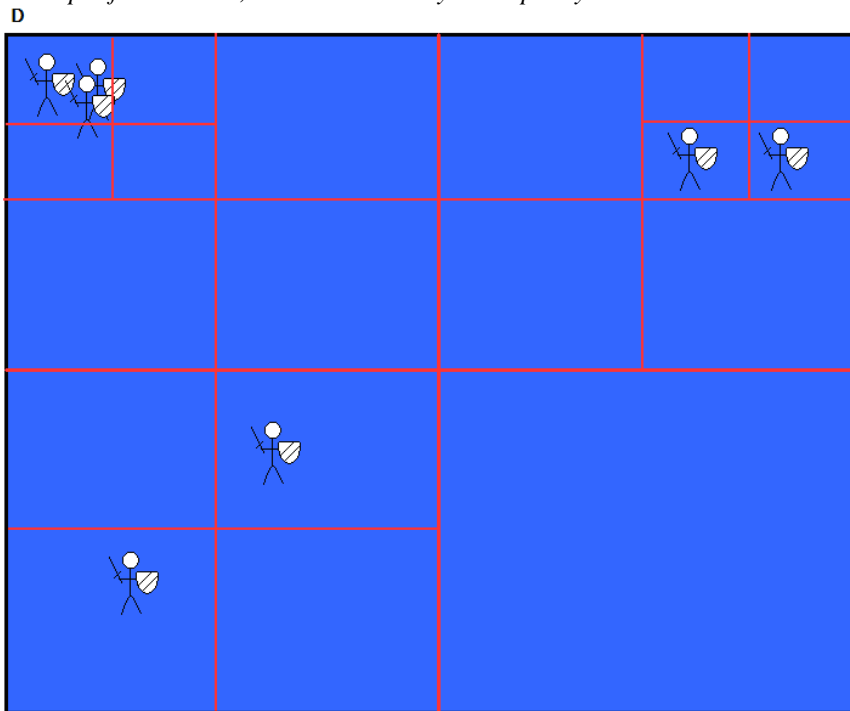


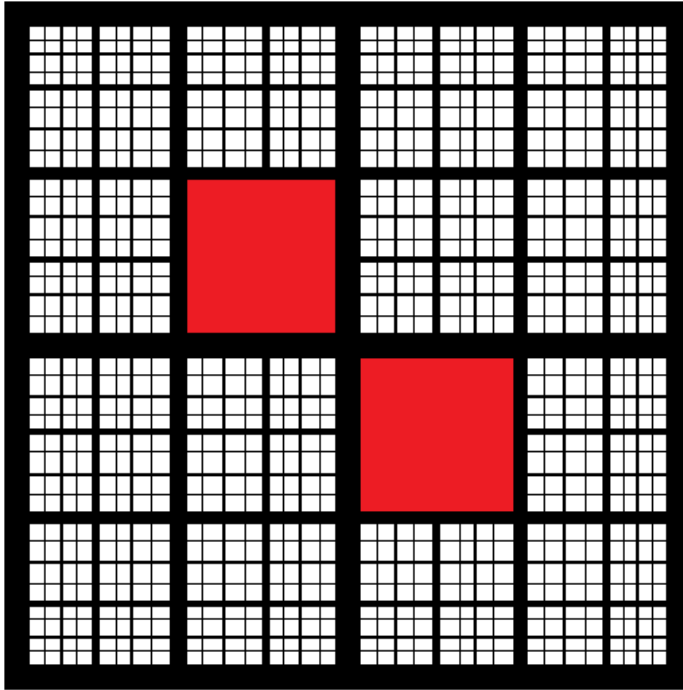
Figure 20: Step D divides the white cells one last time, finishing off the division step entirely. Note that in the top-left cell, there are still three Units. The division ended in this case because in this example we are working with a tree of depth four, and when the maximum depth is reached, a cell cannot be divided any further. So, those three guys in the top-left will have to make do with being a bit crowded for a frame or two.

### ***3.5.4 Active List Construction:***

The function that constructs the Active List is recursive. A simple description of its functionality is to say that it starts at the root of the tree, steps through each cell, and uses the occupant count to determine which cells will make it onto the Active List. Cells with too many occupants must be "broken down" into finer levels, stopping at the appropriate amount of occupancy.

The time complexity of this process is linear, with respect to the number of cells in the Active List. The worst-case time is  $O(N * C)$ , where  $C$  is a constant based off of the worst-case time of finding the neighbors for a single cell, which will of course evaluate to is  $O(N)$ . The term  $C$  is composed of two parts:  $(4(K-3))$ , which is the number representing the total quantity of diagonal same-level neighbors times the quad tree's height at the worst-case level (the second level), and the quantity representing the maximum number of recursions needed to find the active children  $(4(2^{K-3}-1))$ , where  $K$  is equal to the quad tree's height). The second, more interesting of these terms,  $(2^{K-3}-1)$ , can only occur in its worst case at most twice.





*Figure 21: As shown, there can only be a maximum of two worst-case cells (red). Any other positions would not have as many neighbors because the cells would either be adjacent to another worst-case cell, or because they would be adjacent to the edge of the grid space.*

The breakdown is as follows; the  $N$  term exists because the entire Active List is traversed.

The constant  $C$  is used to represent the cost associated with finding the adjacency information for a single cell. The term  $4(2^{K-3}-1)$  is used because, in the worst case, for each of the four cardinal directional same-level neighbors, the function `CheckChildren()` must be recursed starting at two levels below the root of the quad tree and going down to the leaves, at a depth of  $K$ , and at each level, the recursion branches into two calls.

Ignoring the fact that the worst case can only occur to a maximum of twice per Active List, the complexity analysis will proceed using this worst case value as an upper bound for all cells. Because  $4(2^{K-3}-1)$  evaluates to being a constant term with respect to  $N$ , the worst-case time complexity of finding the adjacency information for the entire Active List is  $O(N * (4(K-3)) + (4(2^{K-3}-1)))$ , or  $O(NC)$  where  $C$  equals  $4((K-3) + (2^{K-3}-1))$ , which of

course reduces to  $O(N)$ . It is also worth mentioning that on average, the FindActiveListNeighbors() function finds all of its adjacency information on step (2)A, resulting in a virtually negligible constant  $C$  of exactly eight.

Pseudocode for finding the adjacency information in the Active List:

```

FindActiveListNeighbors(CellList ActiveList)
*START*
For each Cell in ActiveList (n):
(1) SLN[8] := {
ActiveList[n].same-level-neighbor-N ,
ActiveList[n].same-level-neighbor-E ,
ActiveList[n].same-level-neighbor-S ,
ActiveList[n].same-level-neighbor-W ,
ActiveList[n].same-level-neighbor-NE ,
ActiveList[n].same-level-neighbor-NW ,
ActiveList[n].same-level-neighbor-SE ,
ActiveList[n].same-level-neighbor-SW
}
(2) For each Cell in SLN (i):
    A. If SLN[i].isActive == TRUE then
        i. ActiveList[n].AddNeighbor(SLN[i])
        ii. Next i (break and goto 1).
    B. For each Cell in SLN[i].parents (p, from SLN[i] to ROOT):
        i. If SLN[i].parents[p].isActive == TRUE then
            a. ActiveList[n].AddNeighbor(SLN[i])
            b. Next i (break and goto 1).
        ii. Next p.
    C. CheckChildren(ActiveList[n], i)
    D. Next i.
(3) Next n.
*END*

```

```

CheckChildren(Cell cur, int direction)
*START*
(1) If cur does not exist, then RETURN.
(2) Cell children[2] := { FindAdjacentChildrenPair(direction) }
(3) If children[0] does not exist, then RETURN.
(4) If children[0].isActive == TRUE then cur.AddNeighbor(children[0]),
    else CheckChildren(children[0], direction).
(5) If children[0] == children[1], then RETURN.
(6) If children[1].isActive == TRUE then cur.AddNeighbor(children[1]),
    else CheckChildren(children[1], direction).
*END*

```

FindAdjacentChildrenPair(int direction) finds and returns the set of two child Cells that are adjacent to the center cell. There can only be eight possible directions passed into this function, so the function runs in constant time, looking up the correct answers off of a very short list.

### ***3.5.5 Post-Active-List Construction:***

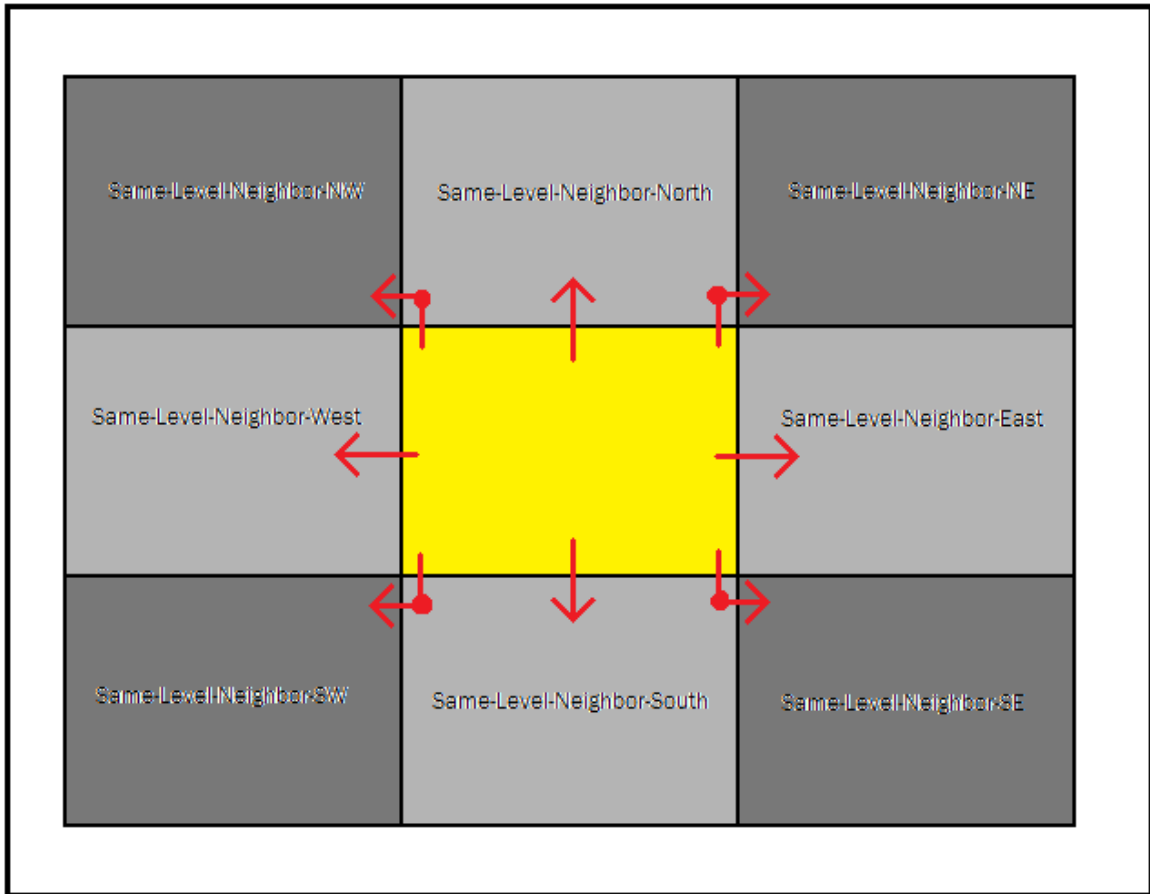
The "post-Active-List" step is the direct opposite of the "pre-Active-List" step, undoing the artificial inflation of various cells' occupant counts. During this step, each cell designated as a goal has its occupant count reduced by the same large constant as was used in the "pre-active-list" step. Going up the tree from each goal toward the root, each subsequent parent cell's occupant count is identically reduced. This should correct the occupant count of all cells, making it accurately reflect the number of Units present on any given cell. In a similar manner, all cells at the designated "minimum level" have their occupant counts reduced and corrected.

Also, during this step, the system iterates through the Unit list in order to determine each Unit's "current cell". Recall that in the pre-Active-List step, each Unit recorded the list of H cells that the Unit occupied. Now, exactly one of those H cells will be Active. That is the cell that will become the Unit's "current cell".

### ***3.6 Step Two: Find Active List Neighbors***

In order to do anything useful with the Active List, we must find all the necessary adjacency information. Therefore, in this step, we iterate through the Active List, figuring out which cells are adjacent to which other cells. The simple description of this process is to consider it a two-step process that is repeated eight times per active cell.

Each cell has four pre-calculated neighbors, called "same-level-neighbors", which were defined during field construction. Go through each one of the same-level neighbors (the four cardinal directions as well as the four secondary directions), look at its parents, then all of its children, searching for any and all Active cells, adding them as an Active List neighbor to the current cell.



*Figure 22: Shortly after the tree's cells are allocated, each level of the tree is iterated through, hooking each cell up with its neighbors on the same level of the tree. During the step where the neighbors for the cells on the Active List are found, these eight same-level-neighbors are used extensively. Each one of the eight is tested; a cell's progeny, ancestry, and even itself, are all tested to find any and all active cells that are adjacent to the center cell.*

### ***3.7 Step Three: Density, Average Velocity, Discomfort***

Once the Active List is constructed and the appropriate adjacency information is found, it can be put to good use.

First, the cells in the Active List are cleared of temporary data. To find the density and the average velocity, iterate through the Unit list. Add the Unit's density and velocity terms to the density and average velocity terms of the Unit's specified "current cell".

Then, iterate through the Active List, dividing the cells' densities by the appropriate cell area to get the appropriate cell density. Also, divide the average velocity term by the number of occupants to find the appropriate average velocity.

Similar to the way density is calculated, discomfort is also found here. Discomfort from a Unit is that Unit's density projected in the direction of that Unit's velocity; it represents where the Unit will probably be in the near future, or at the very least, where the Unit is trying to go. The discomfort from a Unit is distributed in the direction of its velocity. In that direction, a center cell is chosen. Weighted by its distance away from the center cell and all of the center cell's neighbors, the discomfort will be spread among this set of cells.

### ***3.8 Step Four: Find Speed***

To find the speed, step through the Active List, calculating each cell's anisotropic data.

The key pieces of information obtained for any single anisotropic data set are the distance

and the direction of the anisotropy. Once the distance and direction terms are found, a linear interpolation is performed between the maximum Unit speed and the average velocity at the current cell. The parameter for the interpolation is the density at the current cell. At low densities, the maximum Unit speed is favored, and at high densities, the average velocity is favored. When a speed is acquired, clamp it between the maximum and minimum possible speeds (the minimum should be a number slightly above zero). Though there is nothing about the CIG-C technique that prohibits the calculation of terrain contributions to speed, in this paper's tested implementation terrain was left out, choosing to focus only on the crowd's interaction with itself.

*Equation 4: For each cell in the Active List, the speed going in the direction of each one of a cell's neighbors must be individually calculated and stored with the rest of the anisotropic data gathered in this step.*

$D_i$  = The density of the neighboring cell in the direction of  $i$  from the current cell

$A_i$  = Average velocity of the neighboring cell in the direction of  $i$  from the current cell

$dir_i$  = The unit-length vector from the center of the current cell pointing to direction  $i$

$D_M$  = Maximum Density ,  $S_M$  = Top Speed

$$t_i = \frac{D_i}{D_M}$$

$$Speed_i = S_M \left( -t_i \right) + t_i \left( A_i \bullet dir_i \right)$$

### ***3.9 Step Five: Find Unit Cost***

To find the Unit Cost, iterate across the Active List, performing the Unit Cost equation on each cell. The equation makes use of three user-defined weights (alpha, beta, and gamma) in conjunction with a simple speed-based equation. The method used to find each cell's Unit Cost greatly affects crowd behavior. This is where it is decided whether

density, flow speed, or crowd discomfort is weighted more heavily than the other factors. Altering these weights directly changes the behavior of the crowd.

*Equation 5: The Unit Cost function is exactly the same as from the Continuum Crowds approach, except now that our distances between cells is non-uniform, we must reflect that fact inside this equation. Intuitively, just scale the whole term by the distance  $d$ , as shown here.*

$$\int_p C, \text{ where } C \equiv \frac{\alpha f + \beta + \gamma g}{f} d$$

### ***3.10 Step Six: Find Potential***

The potentials at each cell are found using a version of Dijkstra's algorithm. To start off, set the "current cell" to be the goal. Set the current cell's potential to zero, set its candidate flag to false, and set its done flag to true. Now enter the while loop, which continues until the size of the candidate list is zero. Iterate through the current cell's active neighbors list; if a neighbor's done flag equals true, then move on to the next neighbor. Find the anisotropic data that leads from the specified current cell's neighbor to the current cell, which is thusly labeled "correct anisotropy". The projected cost is equal to the current cell's potential plus the correct anisotropy's Unit Cost. If the current cell's neighbor's potential is not greater than the projected cost, then move on to the next active neighbor. If the current cell's neighbor is a candidate, then remove it from the candidate list. Set the current cell's neighbor's potential to the projected cost, then place it on the candidate list. Please note that all insertions into the candidate list should be ordered insertions, from smallest to largest potential. Once all neighbors have been considered, set the current cell equal to the cell at the beginning of the candidate list. Then set the

current cell's done flag to true and remove it from the candidate list. Now, continue with the while loop.



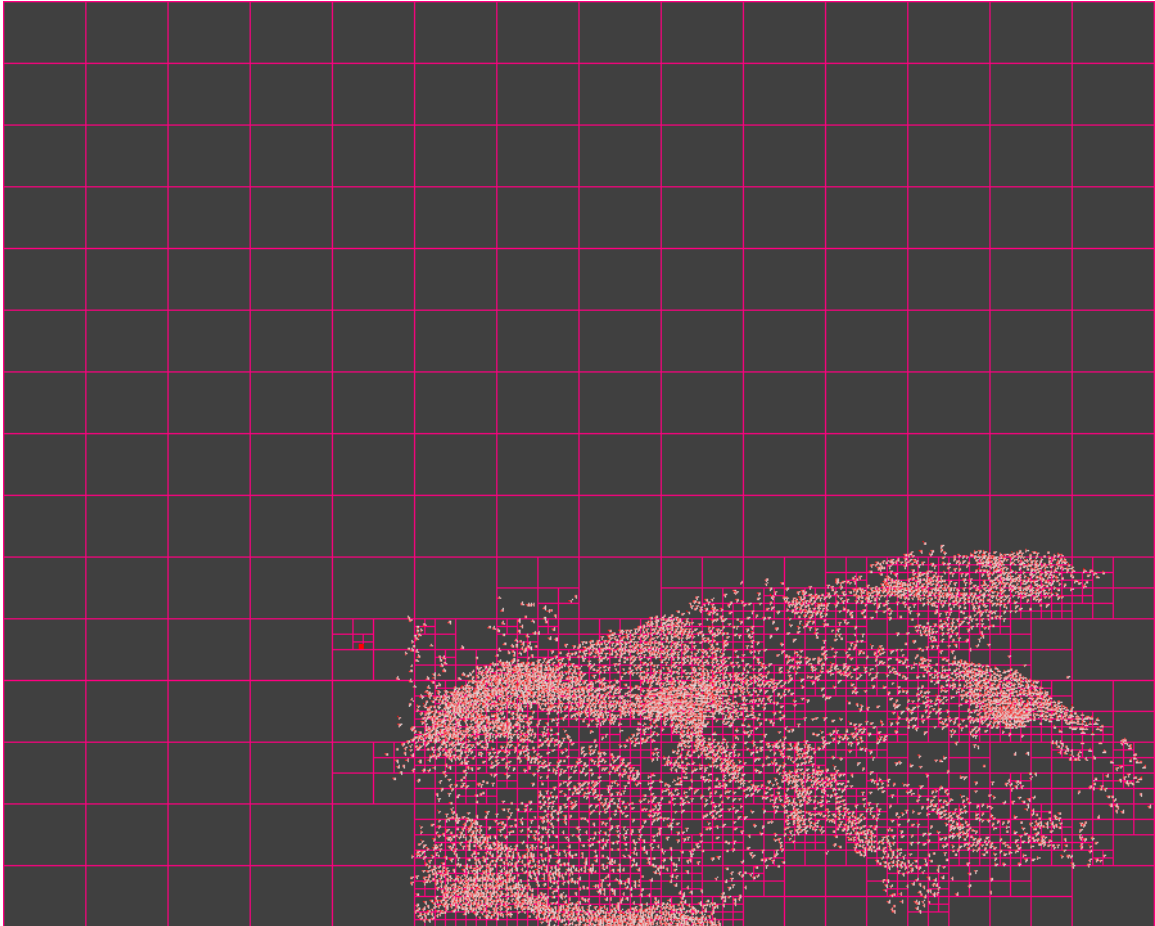
*Figure 23: On a simulation area of dimensions 1024x768 pixels, with a tree depth of 10, at a Unit count of 8000, this is the dynamic potential field for the red goal. The lighter areas represent the places that are easier to get to from the goal, and the darker areas represent locations that are more obstructed or farther away. The Units have been made invisible to better examine the dynamic potential field itself in this picture.*

### ***3.11 Step Seven: Find Final Velocity***

After having found the potential values at each cell, next the final velocities are calculated. Iterate through the Active List, finding the correct direction to travel for each cell. This is found by searching through a cell's list of anisotropic data and finding the piece with the correct potential. The correct potential is the one that, when subtracted by its neighbors potential, is equal to zero. This method of finding the correct potential differs from the method used in Continuum Crowds, where the smallest potential is



always chosen. The final velocity is equal to that anisotropy's direction times that anisotropy's speed.



*Figure 24: Following the final velocity vectors at their respective cells, 8000 tiny Units chase after the red goal as the user moves it around with the mouse.*

### ***3.12 Step Eight: Move the Units***

At this point, all cells' final velocities have been acquired. Now, all that must be done is to move each Unit according to its currently occupied cell's velocity field. However, simply doing so without any other considerations will result in choppy, rigid movement. Therefore, for each Unit, take the weighted average of several nearby velocities. The

velocities to be considered consist of the velocity at the Unit's currently occupied cell along with all of that cell's neighbor's velocities. The weight applied to each velocity is the Euclidian distance from the center of the Unit to the center of each respective cell. Once the averaged velocity is determined, apply it to the Unit.

If desired, a final step, minimum distance reinforcement, can be executed, which is a simple collision check between all adjacent Units. This minimum distance reinforcement insures that no Units overlap each other, though Units automatically avoid areas of high congestion and thus are unlikely to be overlapping other Units.

*Equation 6: Use this finalVelocity value to move the current Unit. The term C represents the current cell, N is equal to the number of neighbors at C, and U is the current Unit. The term C<sub>n→i</sub> translates to mean “the cell that neighbors C in the direction i from C.” The terms V<sub>x</sub> and P<sub>x</sub> refer to the velocity of object x and the position of object x, respectively. The i term of sigma begins at zero rather than one for the convenience of most computer scientists.*

$$finalVelocity = \frac{\frac{V_C}{\|P_C - P_U\|} + \sum_{i=0}^{N-1} \frac{V_{C_{n \rightarrow i}}}{\|P_{C_{n \rightarrow i}} - P_U\|}}{\frac{1}{\|P_C - P_U\|} + \sum_{i=0}^{N-1} \frac{1}{\|P_{C_{n \rightarrow i}} - P_U\|}}$$

## 4 Comparison

Having implemented CIG-C as well as Continuum Crowds, eight test cases were evaluated. The unifying factor in all test cases is that the simulation area always spanned 1024 by 768 pixels, and the maximum number of Units was always 4500. Please note that the CIG-C implementation was set to use a minimum tree depth of four for all tests. The CIG-C implementation was pitted against the Continuum Crowds implementation on a laptop containing the following hardware:

*Toshiba Satellite X205, running Windows Vista Home Premium*

*Intel® Core™2 Duo CPU T7100 @ 1.80GHz / 1.80GHz*

*2046 MB RAM*

*32-bit Operating System*

*Graphics Card: NVIDIA GeForce 8700M GT*

In each test case, the behavior of both implementations was almost identical; the only difference was the performance.

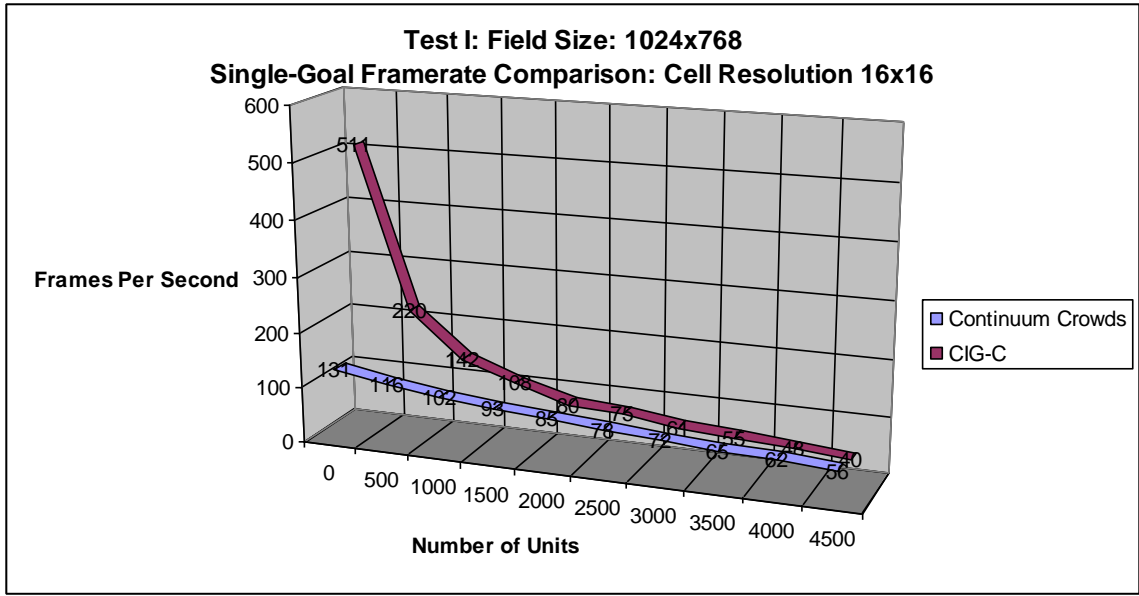


Table 1: To match the resolution of the Continuum Crowds cells (16 by 16 pixels), CIG-C generates a tree with a depth of 7, making the smallest cells 16x12 pixels.

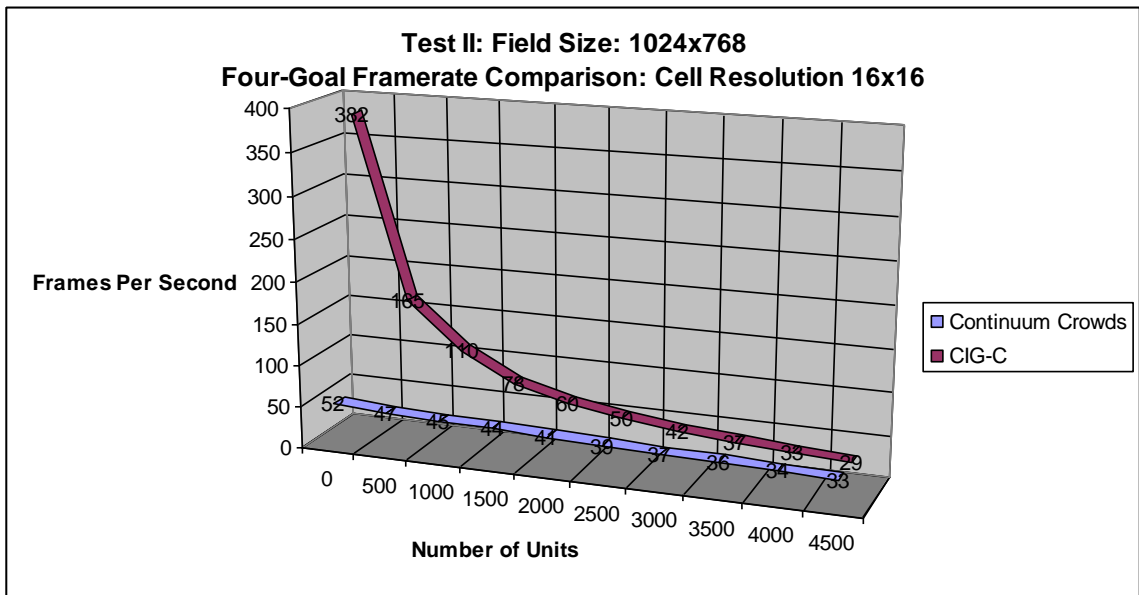


Table 2: As will be seen shortly, Test II marks the last of the test cases where Continuum Crowds even compares with CIG-C's results.

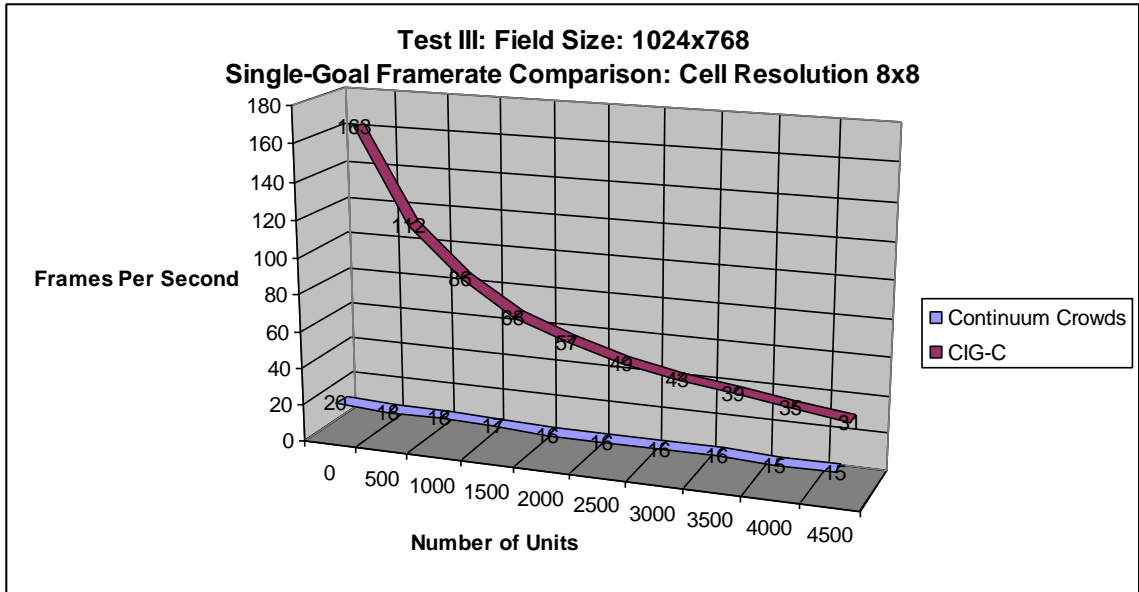


Table 3: To match the resolution of the Continuum Crowds cells (8 by 8 pixels), CIG-C generates a tree with a depth of 8, making the smallest cells 8x6 pixels.

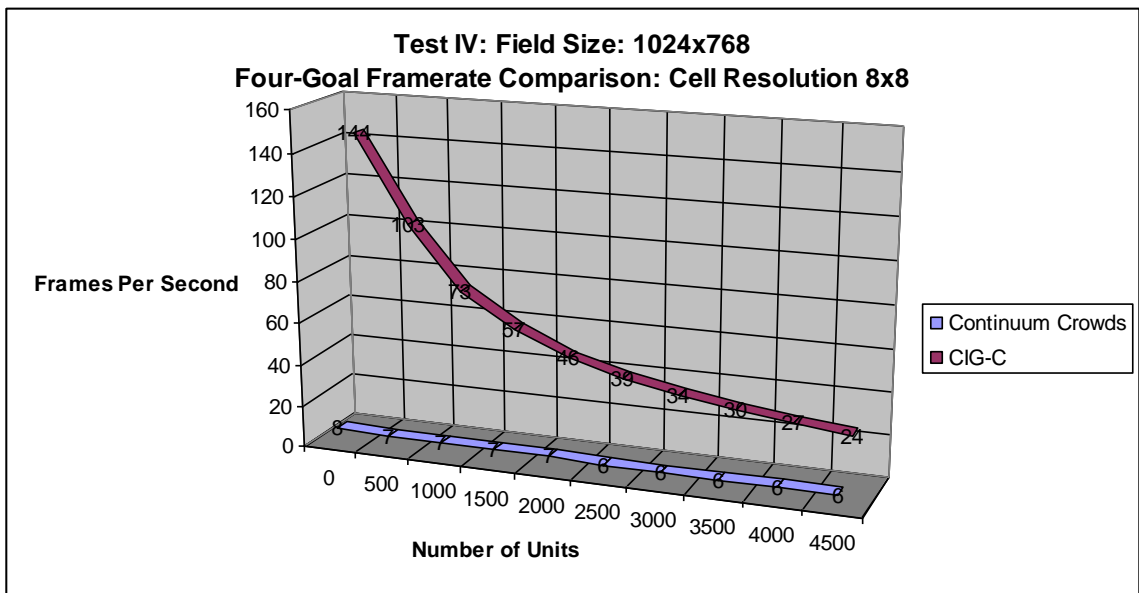


Table 4: At 4500 Units, CIG-C performs four times faster than Continuum Crowds. Also, notice that the number of goals present in the simulation has a greater effect on Continuum Crowds than it does on CIG-C. In Test III, Continuum Crowds ran at 15 frames per second with 4500 Units, but in Test IV, with the addition of three extra goals, the framerate drops by nearly 66%. Compare this to CIG-C, which loses less than 25% of its speed from Test III to Test IV.

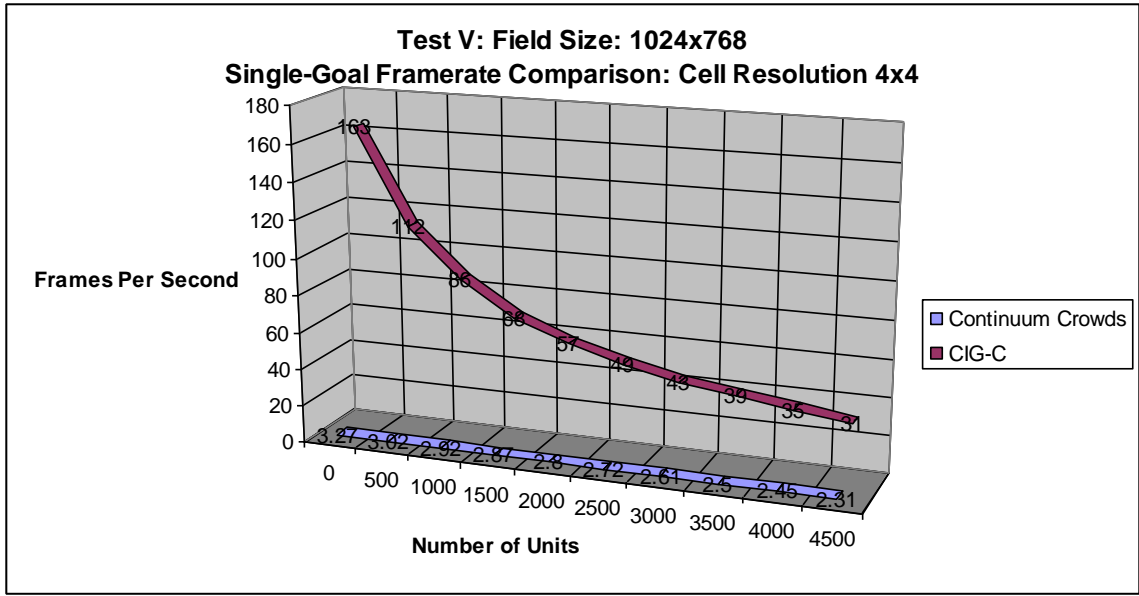


Table 5: To match the resolution of the Continuum Crowds cells (4 by 4 pixels), CIG-C generates a tree with a depth of 9, making the smallest cells 4x3 pixels.

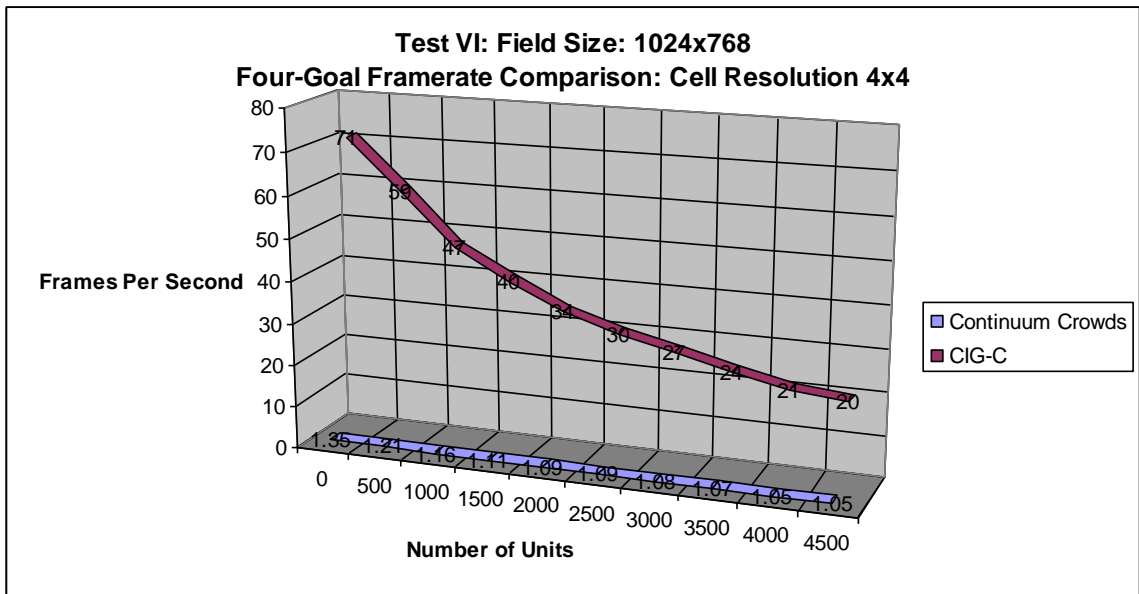


Table 6: As the resolution of the grid increases, the performance gap between CIG-C and Continuum Crowds widens. Here, in Tests V and VI, CIG-C maintains healthy framerates, while the Continuum Crowds implementation begins to lose its “real-time” status.

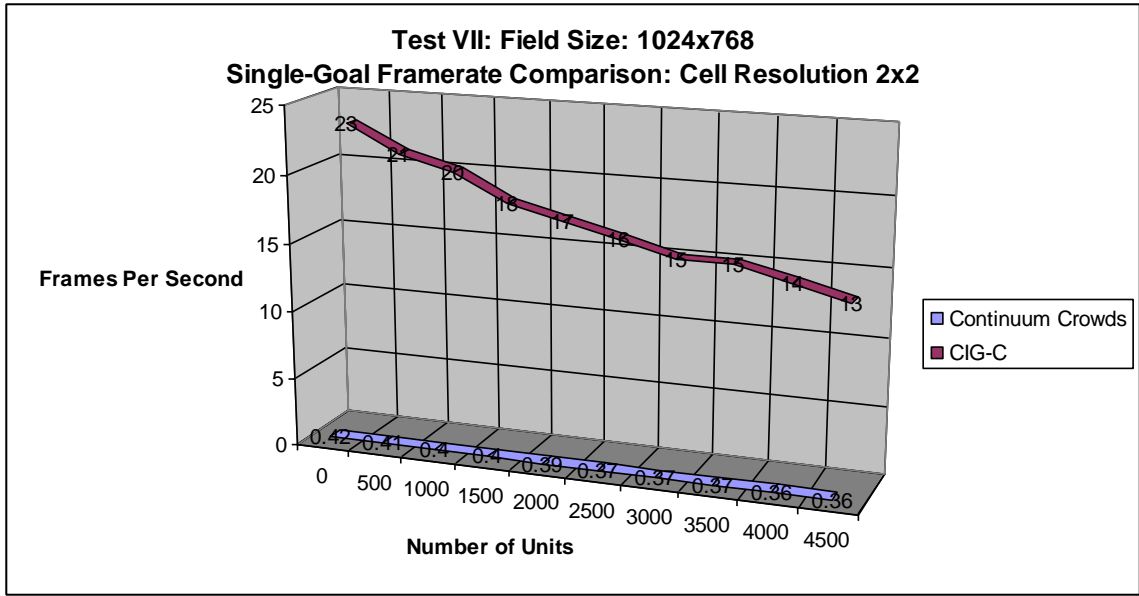


Table 7: To match the resolution of the Continuum Crowds cells (2 by 2 pixels), CIG-C generates a tree with a depth of 10, making the smallest cells 2 by 1.75 pixels.

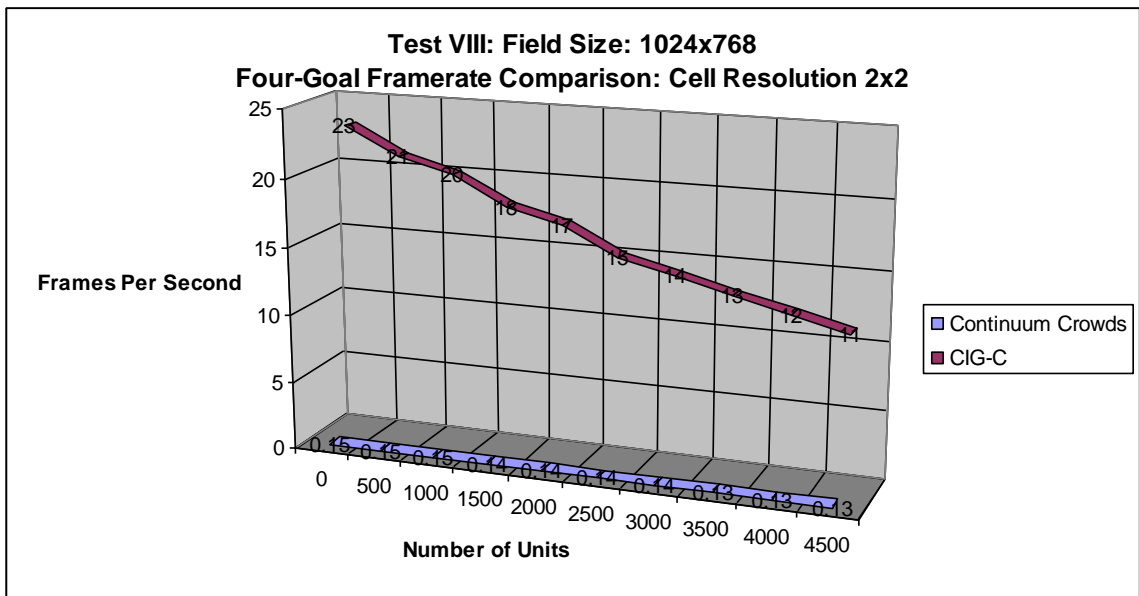


Table 8: Tests VII and VIII push the resolution of the grid to the point where Continuum Crowds takes about seven or eight seconds to complete a single frame of simulation, regardless of Unit count. The CIG-C method continues to flourish, however, maintaining interactive framerates even with 4500 Units and four goals.

From Test III and beyond, the CIG-C method consistently displays a higher framerate than the Continuum Crowds method while maintaining equal, if not superior, behavior.

After the results of Test VIII, one concludes that once the resolution of the cells becomes sufficiently fine, the Continuum Crowds method simply has too many calculations to make; it cannot stay at interactive framerates. By the end of Test VIII, the CIG-C method is over eighty times faster than Continuum Crowds.

The CIG-C implementation was also stress tested. On the same hardware, the CIG-C method was capable of pushing even further, simulating the same size space to a tree depth of eleven, making the minimum cell dimensions smaller than a single pixel (1 by 0.75). There were four goals, and 16,000 Units, with a framerate of three frames per second. The Continuum Crowds implementation virtually froze when similar conditions were tested on its system.



## **5 Conclusion:**

### ***5.1 Results***

The CIG-C method has been shown to be superior in performance to the Continuum Crowds method in situations that demand a very high grid resolution across a very large area. However, CIG-C might not be the right method to use if memory usage is a big concern. At a tree depth of eight, the CIG-C method uses 46 megabytes of RAM, compared to the equivalent Continuum Crowds method at a resolution of 8 by 8 pixels per cell, which uses only 29 megabytes. Though CIG-C offers better performance in this case, it uses over 58% more memory than the Continuum Crowds method. However, it is important to note that CIG-C does not always use more memory than Continuum Crowds. At depth seven, CIG-C has an almost identical memory profile to the equivalent Continuum Crowds implementation at a cell resolution of 16 by 16 pixels.

### ***5.2 Future Works***

If memory is a problem, then a good goal for future works would be to dynamically allocate only the required cells on the Active List each frame without creating a static quad tree. This may be accomplished with the aid of a memory manager to help avoid the overhead of multiple news and deletes each frame. In fact, it is quite possible that such a method could grant large improvements over the current method of tree creation simply by allowing more useful information to remain in low-level caches.

Also, the CIG-C functions that iterate across all Units or all cells in a list lend themselves easily to parallelization. On modern hardware, such parallel processing might yield improvements that could double, even quadruple performance without significantly raising the complexity of the system.

The problem of having too many high-resolution cells across the required grid space is exasperated when the simulation takes on a third dimension. If the purpose is to simulate a crowd across a three-dimensional space, the CIG-C method with a little bit of tweaking would, in theory, be a much better choice than other methods, simply due to its ability to severely limit the number of active cells.

### ***5.3 Possible Sources of Error***

Though extra care was taken to faithfully represent the capabilities of the Continuum Crowds system [8], the claim cannot be made that this paper's test implementation is by any means perfect. In this paper's implementation of [8], there is only a single thread running the entire system, and the GPU is unused for the purposes of path finding calculations. Also, an ordered list is used during Fast March rather than a heap structure. The reasoning behind not including these improvements is that because the exact same improvements could be implemented for CIG-C, comparing both systems with the improvements should be the same as comparing both systems without the improvements. It is hereby acknowledged that this reasoning is a hypothesis, not a fact, and with more time, this reasoning would have been tested.

## 6 References

- [1] Reynolds, C.W. Flocks, herds, and schools: A distributed behavioral model. In Computer Graphics (Proceedings of SIGGRAPH 87), vol. 21, 25-34.
- [2] E. W. Dijkstra: A note on two problems in connexion with graphs. In Numerische Mathematik, 1 (1959), S. 269-271.
- [3] Wikipedia. July 12, 2008. Dijkstra's Algorithm.  
<[http://en.wikipedia.org/wiki/Dijkstra's\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra's_algorithm)>
- [4] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 (2): pp. 100–107.
- [5] Livingstone, Daniel; McDowell, Robert. 2008. Fast Marching and Fast Driving: Combining Off-Line Search and Reactive A.I. <[http://cis.paisley.ac.uk/livingstone\\_mcdowell\\_fmml.pdf](http://cis.paisley.ac.uk/livingstone_mcdowell_fmml.pdf)>
- [6] Sethian, J. A., Proc. Nat. Acad. Sci., A Fast Marching Level Set Method for Monotonically Advancing Fronts, 93, 4, pp. 1591-1595, 1996.  
Dept. of Mathematics, Univ. of California, Berkeley, California.

[7] Sethian, J. A., Dept. of Mathematics, Univ. of California, Berkeley, California,  
November 25, 2006. LEVEL SET METHODS and FAST MARCHING METHODS.

[http://math.berkeley.edu/~sethian/2006/level\\_set.html](http://math.berkeley.edu/~sethian/2006/level_set.html)

[8] Adrien Treuille, Seth Cooper, and Zoran Popovic. Continuum Crowds. ACM  
Transactions on Graphics, 25(3):1160–1168, 2006.

[9] Adrien Treuille, Seth Cooper, and Zoran Popovic. 2008. Crowd Flows: Errata  
Section. <http://grail.cs.washington.edu/projects/crowd-flows/>