A Comparison of Steering Techniques for Virtual Crowd Simulations

Stephen Parsons

A thesis submitted for the degree of
Master of Science
At the University of East Anglia
October 2010

A Comparison of Steering Techniques for Virtual Crowd Simulations

Stephen Parsons

24,800 Words

Abstract

Crowd simulations are implemented across a vast range of applications, from scientific demonstrations, to video games, to films, and as such, the demand for greater realism in their aesthetics and the amount of agents involved, is always growing. A successful real-time Crowd Simulation must find the optimal processing balance between the quality of the environment, the quality of the graphical agent representation, and the intelligence depth of the AI controlling the agents. These choices must be carefully made so that the result is appropriate to the intended context of the simulation. When deciding how best to control the agent, a simulation architect is presented with many possible steering and collision resolution methods to choose from. Many studies present these methods individually, depicting their strengths and weaknesses, but few compare multiple methods in an effort to present the best solution. This thesis attempts to address this by implementing and comparing two popular methods of high level steering path generation, and two low level collision detection methods. These are measured on the merits of their computational efficiency, and their level of realism through user testing, to acquire which combination of low and high level methods complement each other best as an ideal, reusable solution.

Acknowledgements

I would like to thank my supervisor, Professor Andy Day, for his support, suggestions, and wisdom over the last few years. I'd also like to thank my co-supervisor Dr. Stephen Laycock, for his constant assistance, and for reviewing my code and the final work extensively. Both he and Professor Day always managed to make time to meet me on my sporadic days off from work, and without their guidance and help, this thesis would not have been possible.

I am grateful to my parents and grandparents, for their encouragement, and funding of this Masters. Most of all, thank you for always keeping me at it.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Crowd Simulation

With ever increasing computer power available to the modern day programmer, the simulation of virtual agents, both individually and in crowds, is a field where the state of the art is constantly changing. Crowd simulations are implemented across a vast range of applications, from scientific demonstrations to video games, and as such, the demand for greater realism in their aesthetics and the amount of agents involved, is always growing. Crowd simulations contain two main competitors for the allocated processing time; the behaviour and path finding of the crowds, and the graphical representation of the agents. In some simulations, it could be argued that a third aspect can be considered, in that of the crowd's surroundings and environment. The optimal output of a successful simulation is one that can achieve a balanced mix of graphical quality and behaviour, and apply it to the desired number of agents.

To move agents in a crowd simulation, a steering algorithm is implemented that suits the requirements of the scene to be navigated. This is a set of rules and decisions laid out by the designer that describes how an agent is to move from one location to another and what to interact with along the way, if necessary. These concepts of large scale movement and local interactions are described as high level and low level steering, and will form the basis of the comparison described in this thesis. High level steering typically implements a path finding system to move an agent around a scene, avoiding impassable terrain or obstacles. A high level steering algorithm is aware of the scene on a macro scale, and is tasked with giving the agent purpose in its motion. Low level steering methods are intended to stipulate how an agent handles interactions with other agents, and how to deal with interruptions from their intended route. Commonly, a method of collision detection is implemented, with a number of outcomes defined as resolutions depending on factors such as the direction, speed, or type of interaction encountered.

The aim of any simulation is to offer a sufficient level of realism, either visual, behavioural or both, that fulfils the purpose of that scenario. Evacuation simulations rely heavily on the behaviour of the agents being as accurate as possible, while the population of a virtual heritage site might concentrate more on the visual quality of the inhabitants, as a trade off for density or intelligence. Any crowd representation that does not focus on the fundamental requirements of the application to which it is to be applied, risks its usefulness becoming nominal. Humans are able to view any virtual representation of movement, from flocks of birds or the movement of groups of agents, and gauge from

personal experience if what they are seeing appears realistic looking. It is the task of the researcher to identify what rules define the realism in the chosen type of movement, and apply that multiple times across all of the objects involved. In many cases, the context of the simulation defines the expected behaviour of the individuals involved, and therefore what the person viewing the scene subconsciously expects to see. In one of the first commercial uses of agent flocking, Disney created a stampede of wildebeest using computer animation driven by artificial intelligence (AI) that was based on Reynolds flocking model [Reynolds, 1987]. This was used in their animated film 'The Lion King'. The diversity of the different wildebeest animation used was limited, however due to the scale of the scene, and the way in which the animals moved, the viewer is successfully satisfied that the stampede looks realistic.

The majority of published studies on crowd simulation aim to focus on adapting an existing steering algorithm or method, and perfect it for use with the intended scenario. Studies that emphasise intelligence over graphical rendering may also amalgamate a low level steering technique with a high level method, but few compare a selection of steering methods. This thesis will present a comparison of two popularly implemented low level steering methods, coupled with two high level path creation techniques. A Voronoi Graph and a Visibility Matrix, both navigated by Dijkstra's shortest path algorithm, will represent the high level methods of scene navigation. A grid based Cellular Automaton Model and a proximity testing method will be compared as competing low level steering techniques. Research into these methods is described in greater detail in the planning and design chapters, with implementation explained subsequently. Finally, each combination of methods are measured on their computational complexity at runtime, their ability to be scaled to larger and more densely populated crowds, and their level of aesthetic realism, through user testing. Conclusions and findings are presented at the end of the thesis.

1.2 Objectives

To be able to measure the success of each approach presented in this thesis, the choice of simulation methods needs to be implemented in a comparable fashion, and in a similar environment. The focus will be on the steering of agents, with environment and agent representation as secondary considerations, to allow maximum processing to be devoted to the steering methods. While moving the agents from point to point is a fairly simple exercise in itself, the difference between each implemented method will be how it deals with agent to agent interaction and the avoidance of obstacles. For this thesis to be effective, this needs to be a primary objective for each steering method. Additionally, each method will require the same amount of agents and period of runtime; therefore each

simulation must be designed in a way that can be run for an indefinite amount of time, to allow for fair and correct analysis.

It is important that the simulation attempts to recreate steering methods specified in existing publications as accurately as possible. The aim is to implement the chosen steering methods within the scale of the designated environment, and therefore each method needs to be applicable to the size of the implemented scene. Hopefully, by fixing such factors, each method will be sufficiently diverse to make for clear results when comparing their effectiveness. It is from this analysis that a conclusion can be drawn to propose which combination of low and high level methods are most effective, or which components could be combined to create an optimised solution.

While it is beyond the scope of this project to create a 'lifelike' representation of human movement, its objective is to present the best possible implementation as put forward by other publications, to which the grounds for such an objective could be built, on the hardware available, and in real-time.

1.3 Glossary of Terms

Throughout this thesis, these terms are used, and are listed here with appropriate definitions.

Agent – A virtual member of a simulation, normally the representation of a person.
AI – Artificial Intelligence.
Boid – A "bird-like object", as defined by Craig Reynolds in [Reynolds, 1987]. Commonly used to reference members of a flock or group of agents.
CUDA – Compute Unified Device Architecture. Nvidia's programming language for controlling calculations on their graphics cards.
FOV – Field of view.
FPS – Frames per Second. Used for measuring the frame rate of graphics rendering.
Game Engine – The combination of rendering, physics, AI, and system code, which together produces a foundation for a graphical program.
GLSL – OpenGL Shading Language.
GPU – Graphical Processing Unit. The chip found on a video card for processing graphical data. This is usually accompanied by VRAM.
LOD – Level of detail.
LOS – Line of sight.

Texture – A graphical representation that is drawn on top of a graphical model to alter its appearance. This is commonly used in building modelling to make a single structural design look like many different buildings.

VRAM – High speed RAM that is found on a video card and is used as storage for the GPU's processes. It also holds geometrical data such as textures, and the framebuffer.

2 Crowd Steering

This chapter covers an overview of what components make a crowd simulation, and explores each one in detail. The latter parts of the chapter more formally explore the main subject of this thesis, looking deeply into steering methods and the algorithms behind them. This contains research and observations gathered from various existing publications.

2.1 Agents

Agents are the population of a virtual simulation which form together to make crowds. They can take any given form, such as a 3D representation of a human, or a more simplistic representation in 2D. How the agent is drawn is determined by the requirements of the scenario. Broadly speaking, the quality of the agents is determined by the focus of the simulation and the computational power available (assuming it is real-time). For example, evacuation and disaster modelling simulations focus very heavily on path finding and physiological effects on the agents, and as a consequence, de-prioritise the quality of the agent models, normally settling for basic 3D representation, or even a 2D equivalent using a top down viewpoint. In Figure 1, an example of this is shown with a scene consisting of flat 2D buildings, and populated by many agents represented with simple green arrows. In contrast, urban simulations focus more on realism of the city and the inhabitants, leading to more animated agents represented in higher polygon counts. Figure 2 shows an example of high quality human models used in this fashion. In this context, their aim is to help create a more realistic looking scene, as shown by the quality of the accompanying buildings. Further detail can be administered to agents in a 3D scene by utilizing a variety of animations. Many pedestrian simulations disregard differences among individuals that make up a scene and use the same animation for all agents. This is generally true for large scale simulations in a move to lessen the computational complexity of running in real-time. Smaller, more detailed scenarios or those with plenty of processing power available, may implement a wide variety of animations to give the members of the scene a more lifelike and individual appearance. Again, this depends entirely on the purpose of the simulation and objectives of the designers. Simulations that utilise a selection of individual animations often use environmental or circumstantial triggers to make agents perform these animations. Examples in an urban environment include pedestrians stopping to look at shop windows, or picking up objects from the floor. Such events must be implemented with the correct frequency, proportional to the density of the agents, to reflect the likelihood of these actions in the real world.

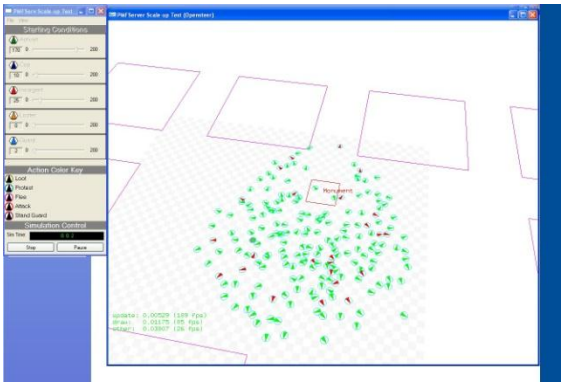

Figure 1. Symbolic representation of agents is often used when visual quality is not as important as the AI of agents, thus saving resources.

Figure 2. High quality representations of humans, seen in a virtual city simulation.

Most 3D agents will have their models covered with a texture to vary their appearance without needing to redesign their structure. The correct variety of textures evenly distributed within a crowd can give the illusion that every agent is individual, but with the added benefit of using very low computer resources (comparatively to extra animations, or different character models). As the quality of a texture increases, so too does the texture memory that is required to store the texture in the VRAM, so a simulation designer must find a balance between quality and resources.

Detailed textures can give a relatively simple agent model the look of a high polygon equivalent. For example, some of the agents shown in Figure 2 are seen to be wearing suits and shirts. A graphical artist that wished to create a very high detail model for a film or single artistic render might approach each item of clothing separately to the person's model, and create the suit jacket and trousers entirely in their own 3D structure. These would then be placed around the human model in a like for like fashion (comparing the real world to the virtual equivalent). This approach is ideal when one wishes to control the intricacies of an agent and his attire, but for a crowd simulation, this amount of processing would be crippling when used for many agents. Instead, agent designers for simulations commonly create a simple character base model and, use a texture to draw on what is not there in the structure. The problem with this approach is that it not possible for the clothes to behave as clothes; as they cannot bend or crease as the human moves [Ryder, 2005]. By using textures, physical attributes such as facial features where the nose and ears would require many extra polygons to model, can be represented with far less computational expense.

Further to this, many simulations utilize a LOD mechanic that increases or decreases the quality of the agents in relation to the viewpoint of the user. Multiple representations of the same agent type are prepared and then swapped in, as the distance from the viewpoint

increases. The theory is that as the distance increases, the user will not notice the less detailed character model or texture, therefore maintaining little noticeable degradation in overall viewing quality in the simulation. Techniques such as alpha blending help to merge the higher and lower representations together, in an attempt to swap them seamlessly when a transition is necessary. D. Pearce et al. [Pearce, 2004] combine LOD with view frustum culling, algorithmically removing agent geometry before the graphics pipeline can apply Hidden Surface Removal to the scene. LOD methods can also be applied to animations, with some simulations reducing the quality and frequency of complex animations when the agent is at an arbitrary distance from the viewer. Silva et al. [Silva, 2008] use a similar method to [Pearce, 2004], but also exclude AI flocking calculations outside of the agent LOS. This improves the Boids model presented by Reynolds in [Reynolds, 1987] by up to three times, using the same methodology, but with occlusion applied.

2.2 Virtual Environments

The virtual environment is the realm in which simulated agents navigate, giving them context and purpose. Similarly to the agents themselves, it can be represented on the most basic level in 2D, or as a far more complex example, such as a photo realistic city. Depending on the purpose of the simulation, the virtual environment can either complement the agents within it, or use agents to complement itself.

As previously described, the virtual environment is one of the three main consumers of computational resources in a simulation. Design parameters such as draw distance and LOD factor greatly into how detailed a virtual environment can be with the resources available. Some designers [Low, 2007] choose pre-made environments for their simulations, such as the freely available Unreal Tournament 2 (UT2) game engine. These are supplied with the environment, lighting, and agent quality already in balance, allowing a scene to be engineered as required far more rapidly. An example of this is shown by Low et al. [Low, 2007], where their crowd AI uses the UT2 engine for visualisation. In the same way as an agent is made to look more complicated than its 3D model actually is; virtual environments utilize textures greatly. For example; a simple square polygon can be made to appear as a row of shops with the application of an appropriate texture. LOD and FOV methods allow fewer resources to be used on buildings that are far away from the viewer, either reducing their render quality, or culling the buildings completely from the viewpoint at an appropriate position in relation to the camera. When this concept is expanded and improved, entire cities can be generated that closely represent their real world counterparts. Figure 3 is a large scale example of this, showing multiple buildings

and individual textures. The quality of such simulations is sometimes in excess of what can be rendered by the available hardware. In such circumstances, footage of the scene may need to be created ‘offline’ by processing pre-rendered images.

Figure 3. A very detailed simulation of a city displaying the impact that high quality textures can have on appearance.

Building modelling can be taken much further to the extent of using satellite data to generate a cityscape. Flight simulations are most common place for such implementation, with Ubisoft’s ‘H.A.W.X’ using GeoEye’s commercial-use satellite imagery to accurately depict the look, structure, and height of buildings in major cities. Figure 4 shows Rio de Janeiro represented within this game, as a prime example of this technique. Although the wide scale would not be appropriate for most crowd simulations, the principle can be scaled to suit.

Figure 4. Rio de Janeiro, represented in Ubisoft's 'H.A.W.X', is an impressive example of satellite topography used for cityscape mapping.

The environment that agents populate is not solely for the purpose of aesthetics. Many demonstrations utilize aspects of the virtual environment as focus points for the agents, triggering behavioural responses. An example used earlier was that of an agent stopping to view a shop window, therefore adding implied realism to the scene.

2.3 Steering

Steering is the control behind the movement of agents in a virtual environment. It is the process that defines their decisions for collision avoidance and path finding, with the aim of simulating that of their human counterparts. Defining behaviour in scientific terms is incredibly difficult, and consequently, there are many different approaches that have been presented for the steering of agents. A steering method is designed or chosen based on the requirements of that simulation and will aim to capture the essence of the human behaviour perceived appropriate in the given environment.

2.3.1 High Level Steering

Path finding is the highest level of steering and controls the agent's routing from one point to another. Routing methods can be based on pre-computed locations or calculated on the fly, depending on the type of path finding. Some pre-computed examples follow a routing table that defines the relationship between predetermined points in an environment, and can be used to calculate the navigation required to get from one point to another via the shortest path. The routing tables, or graphs, used in many steering methods will often be computed using Voronoi graphs, Delaunay Triangulation, or a

Visibility Matrix [Hoff, 2000], [Choset, 1995], [Arikan, 2001]. Other methods of path finding choose a target for the agent to reach and attempt to navigate him there in the most efficient way, while keeping to an 'ideal path'. Methods such as this put a greater emphasis on collision avoidance, requiring checks for both fixed and moving obstacles. The advantage of using pre-computed routing data generated from static obstacles (such as buildings) at pre-processor time, is that these obstacles are already accounted for before runtime, and factored into the accessibility of routes in the scene. This lowers the stress upon low level steering techniques, requiring them only to deal with the avoidance of dynamic obstacles such as other agents.

2.3.1.1 Voronoi Tessellation

The Voronoi diagram is a method used to decompose the space *S* into regions around each point *P* named as the *Voronoi site*, such as all the points in the region around point *P* are closer to *P* than any other point in the space *S* [Champagne, 2005]. Voronoi diagrams are used to divide up an area containing obstacles to avoid, known as Sites, with lines that are as far as possible from each of these points. These lines are called Edges, and are very useful in the context of a crowd simulation as they can be used as paths to navigate a scene, while keeping a maximum distance from surrounding obstacles. The points that join the path edges are known as Nodes, and can be conveniently used as waypoints for agents to navigate to. Figure 5 shows a section of a Voronoi graph, with these key elements highlighted. The implementation described in this thesis will aim to use the centre of each building as the Voronoi site, meaning that the graph will generate paths furthest from each building. This will be described in more detail during Chapter 5.

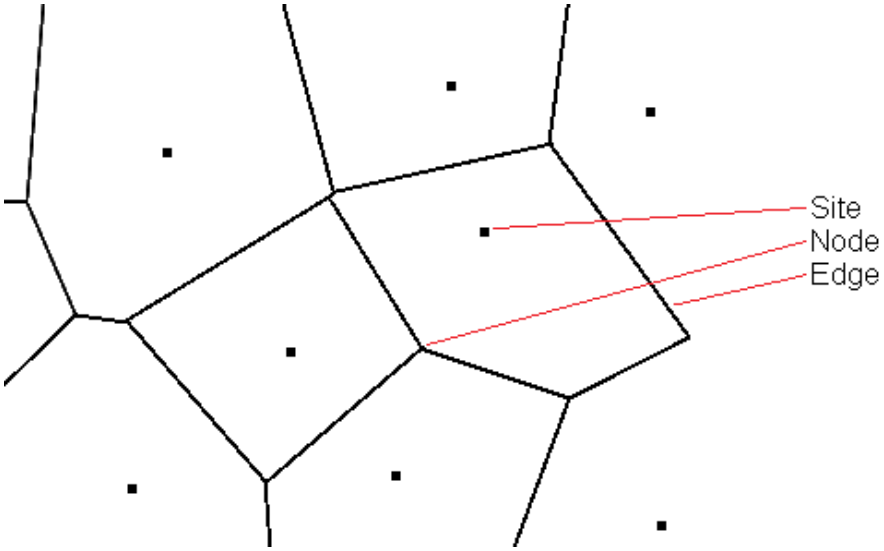

Figure 5. Voronoi diagram. Displaying Sites as the points from which a Voronoi graph is calculated, Nodes as the intersections between Voronoi regions, and Edges as the boundaries of these regions.

Voronoi diagrams are commonly used for agent steering in various presentations for global path finding and goal seeking, although the ways in which they are implemented can vary somewhat. A majority of implementations use Voronoi segments calculated from static obstacles to compute safe paths of maximal clearance for navigation [Hoff, 2000], [Choset, 1995]. In these examples, Voronoi diagrams are created based on a fixed set of points around the environment, and navigated with a path finding method. Hoff et al. [Hoff, 2000] choose an end point in the environment for a robot to navigate to. The closest Voronoi node in the graph to the robot's current position is determined, and then moved towards using a potential-field method. Once on the Voronoi graph, the robot is moved between each node via the shortest graph edge, until the target is reached, although it is not specified which path finding algorithm is used for this purpose. The potential-field approach for governing movement imposes a repulsive force upon the robot or agent calculated based on the distance from surrounding obstacles. The closer the obstacle, the greater the force towards an opposite direction is. In Choset et al. [Choset, 1995], the Voronoi graph is created on an ad hoc basis by exploring an environment with a robot. This robot is equipped with a sonar ring consisting of multiple sensors which can return the distances of obstacles in the robot's vicinity. It moves around an unmapped scene using the algorithmic rules that define a Voronoi graph, maintaining equidistance from all obstacles within its line of sight. The graph produced is a robust representation of a static scene Voronoi graph, and with modification, the algorithm used to construct the graph could be adapted for procedural generation within a program, instead of using a sonar equipped robot.

In contrast, Champagne et al. [Champagne, 2005] compute Voronoi diagrams based around the centre of each group of agents, using these regions to determine the boundaries the members of that group can populate. This causes varied degrees of agent flocking, ranging from tight to sparse, depending on how small the region allocated to that group is at that given time. In this simulation, each group of agents moves around a scene using a FOV of 120 degrees, checking for possible collisions at 3 set distances in front of this FOV. Collisions with other flocks are avoided at long distance by adjusting the direction of both groups to a trajectory parallel to the tangent of the intersect point of the bounding circles. This keeps the groups moving, while maintaining sufficient distance between them to allow free movement of the individual agents. Pending collisions with several groups, or collisions that are detected close by, can provoke the flock to adjust their speed or even stop completely until the conditions are met that allow them to proceed. The ability for a crowd to halt entirely is often overlooked or not implemented in a majority of simulations. Similarly, A.Sud et al. [Sud, 2007] use dynamic first and second order Voronoi diagrams to create the paths for virtual agents to follow. These are

calculated per frame by their MaNG Engine (Multi-agent Navigation Graph), but in contrast to the method used in [Champagne, 2005], the MaNG is calculated on the GPU, allowing for greater performance due to faster processing.

2.3.1.2 Visibility Matrix

Another method for dividing up a given space for use by a path finding algorithm, is to use a Visibility Matrix. This requires the space to be defined by the vertices of the obstacles within it, each representing a node within the Visibility Matrix. The relationship of each node with every other node in the matrix is tested to measure whether a path can be drawn between them without intersecting an obstacle. The results of this are recorded within the matrix, creating an array detailing the paths between every pair of nodes. To enable common path finding algorithms to be run against this matrix, the lengths of each successful path can also be recorded. The strength of the Visibility Matrix is that it can be run against any free space with fixed obstacles defined by simple polygons, and create paths without requiring manual selection of path nodes, or any further intervention. Additionally, it can be used as a pre-computed matrix, so the complexity or number of nodes does not affect the runtime performance. Arikan et al. [Arikan, 2001] use a Visibility Matrix constructed in this way. This study computes the shortest paths between each pair of vertices at compile time, and navigates the matrix at runtime using Dijkstra's shortest path algorithm. Figure 6 shows a diagram used to display this method. Their reasoning for using a Visibility Matrix for path creation is that it does not involve segmenting the environment with landmarks, and therefore, does not suffer from issues such as local abnormalities which may be created from procedurally generated landmarks or waypoints. In contrast, Byszewski [Byszewski, 2009] uses waypoints set manually within a scene editor, allowing for full customisation. This does not, however, offer the speed and Automaton required by a majority of studies, nor by the aims of this simulation. The requirement for steering agents through paths is generally due to the need to avoid obstacles, which the Visibility Matrix method does in a logical fashion by using the edges of the obstacles themselves as waypoints.

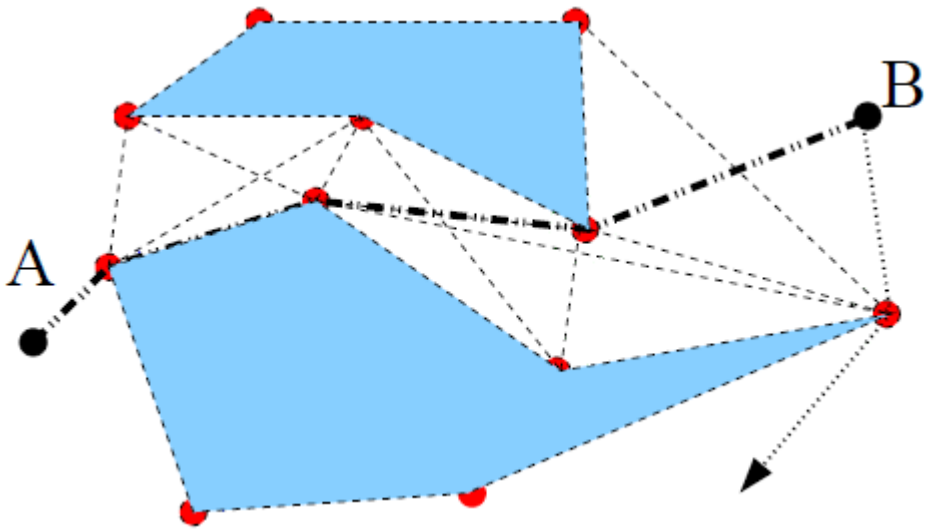

Figure 6. A visibility graph constructed from the nodes shown in red, and then navigated from A to B via the shortest path. Adapted from [Arikan, 2001].

Mitchell [Mitchell, 2000], and Hershberger [Hershberger, 1989], both describe methods of extending the edges of polygons created by Visibility Matrix algorithms, if certain circumstances arise. Examples arise at scene edges where large open spaces can occur due to vertices only being present along the borders of more centrally located obstacles. By extending path edges outwardly from outlying obstacle edges, to meet the bounding scene border, the previously open space is dissected into smaller triangular sections. These are both easier to navigate and reduce the chance of bottlenecking by offering fewer paths towards the same goals.

2.3.1.3 Dijkstra's Algorithm

Dijkstra's path finding algorithm is used to find the path with the lowest cost between a given node in a graph, and every other node in that graph. This is known as finding the shortest path. It is normally applied to a graph or tree consisting of multiple nodes and the path distances between each node, known as the path weight. The algorithm expands outwards from a starting node, navigating through the graph, searching for the next shortest path, until its target is located. The list of nodes traversed towards the target can then be read back as a complete route, and used as the optimal method of reaching the given destination. The algorithm's procedure can be best described as follows (adapted from [Dijkstra, 2011]):

1. Set the initial source node a distance value of zero, and assign every node an infinite distance.
2. Mark all nodes as unvisited. Set initial node as the current node.

3. For the current node, consider all its unvisited neighbours and calculate their distances from the initial node. For example: if the current node *x* has a distance of 8, and an edge connecting it with another node *y* is 5, the distance from *y* to *x* will be 8+5=13. If this distance is less than the previously recorded distance from the initial node, overwrite the current route distance.
4. When we are done considering all neighbours of the current node, mark it as visited. A visited node will not be checked again; its distance recorded now is final and minimal.
5. If all nodes have been visited, or if the target node is reached, finish the algorithm. Otherwise, set the unvisited node with the smallest distance from the initial node, as the next current node and continue from step 3.

Figure 7 shows two identical layouts and their waypoints, as interpreted by a Voronoi Graph and a Visibility Matrix. The first two diagrams display the entire graph, with example path distances. These are displayed on the Voronoi, but for ease of display, are omitted from the Visibility graph. The second pair of diagrams display the same scenes but with two paths chosen by Dijkstra shortest path algorithm, displaying how the algorithm described above, decides upon the final route.

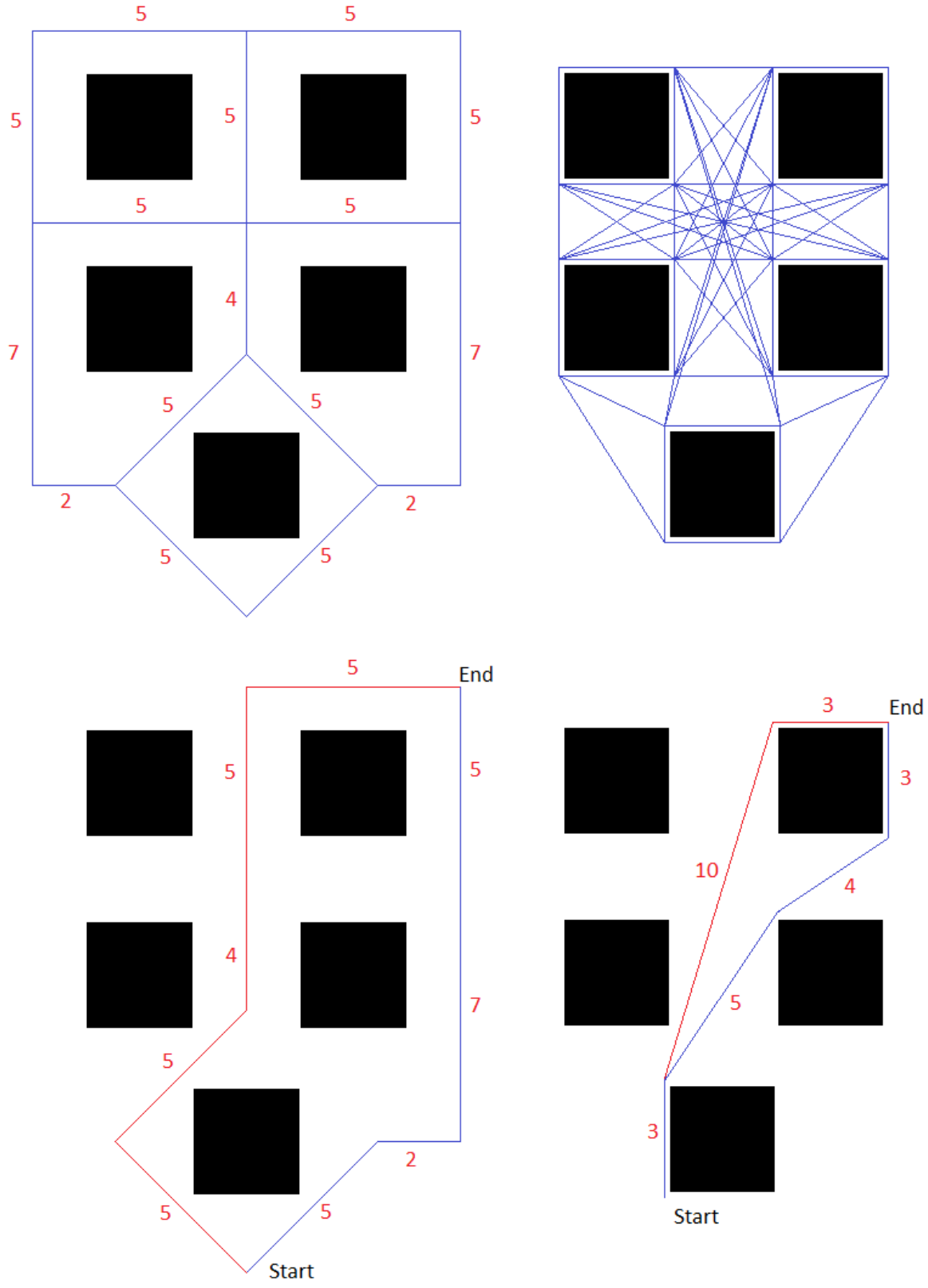

Figure 7. This diagram shows how Dijkstra shortest path algorithm chooses a path from a waypoint matrix created by a Voronoi Graph or a Visibility matrix, based on the cost of each waypoint. Shortest paths are shown in blue, with a longer red path given as a contrasting example.

Dijkstra's algorithm is commonly used in crowd simulations for agent path finding due to its ease of implementation and the accuracy of its output. As described previously, Arikan

et al. [Arikan, 2001] use Dijkstra's algorithm against a Visibility Matrix for agent routing. In this study, all possible paths are pre-computed before runtime and stored within an array, to be called upon when an agent requires a new path. Using this method, the runtime overheads of path finding are negligible as Dijkstra's algorithm is never run again. The problem with using stored routes is that it does not allow for dynamic route changing, such as that seen in Pelechano et al. [Pelechano, 2007]. This study concerns evacuation simulations, and proposes circumstances where a path may become permanently blocked or temporarily impassable. Such situations could not be dealt with in a simulation that uses only precompiled routes from each node. Haciomeroglu [Haciomeroglu, 2009] uses two versions of Dijkstra's shortest path finding algorithm at multiple stages when navigating agents. The first is for global path calculation on a city wide, free space map, and the second is to navigate sub-paths between global paths. At points where an agent enters a large area of free space between two global nodes, an agent may be presented with the possibility of reaching the next global node via several sub-paths. This is where the second Dijkstra's algorithm is applied, and navigates using the current population density of each sub-path to represent the path weighting. This evens out the population preventing overcrowding. Mitchell [Mitchell, 2000] describes a variation of the standard shortest path algorithm called the Continuous Dijkstra Method. In this, he describes a technique that propagates wave fronts equidistant from a source, that span out to discover obstacle vertices. This essentially builds a visibility graph on the fly while seeking the target node, and is effective in scenarios where the scene cannot be first mapped at pre-computation time.

2.3.2 Low Level Steering

As with global path finding techniques, there are many methods of collision avoidance that can be used to aid the navigation of the agent around a scene. These are categorised as low level steering techniques, and can be complementary when paired appropriately to certain high level counterparts. The complexity of the collision avoidance required depends on how comprehensive the path finding may be. Pre-routed path finding methods may only require the agent to be aware of, and avoid, other agents; while on the fly routing will require a heavy reliance on collision detection and avoidance to navigate the agent around both agents and obstacles.

2.3.2.1 Grid Methodologies

One approach to dealing with inter-agent relationships is to divide a given space into equally sized cells, basing AI decision on the state of each cell when populated by an agent. This is known as a Cellular Automaton Model, and is described in a study by

Dijkstra et al. [Dijkstra, 2000]. This method of scene subdivision models a physical system as a mathematical representation, assigning each cell within it a finite set of states. Traditionally these consist of two states, either populated or empty, and can be used to represent both static and dynamic obstacles, depending on the desired application. Figure 8 shows a building represented in a Cellular Automaton Model.

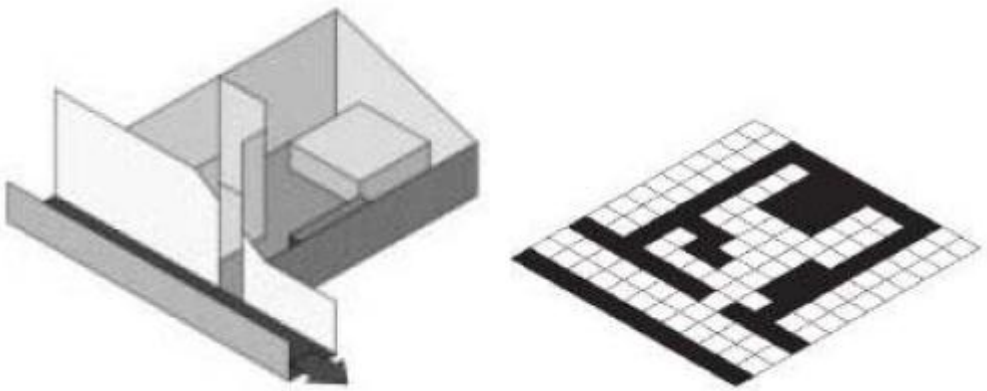

Figure 8. A structure represented by a Cellular Automaton Model, with black squares signifying impassable grids. Adapted from [Pelechano, 2007].

A Cellular Automaton Model will use a fixed resolution to interpret the scene, which will be specified by the size of the uniform grid of cells applied to the space. As agents navigate a scene, their presence within the grid cell must be tracked and updated, allowing the global grid to maintain a track of all obstacles contained within it. This offers a favourable ability to check for collisions between agents with very low computational overheads, as the agent need only request the status of a grid square from the global grid. As seen in other methods, a proximity check would normally be conducted to discover neighbouring agents or potential future collisions. Assuming an agent is not navigating along the edge of a scene, there are only eight potential directions an agent can take from a given location, further reducing the computational complexities associated to other navigation methods. However, this factor, and others present potential disadvantages with the Cellular Automaton Model as observed by in Pelechano et al. [Pelechano, 2007]. Due to the resolution of the cells within the grid, the densities allowed within each area of a given space are usually less than that which would be allowed in real equivalents. This is apparent when a section of geometry or an agent overlaps multiple grid cells. If all partially populated grid cells are marked as occupied, the space allotted to that agent or obstacle is likely to be much greater than that of its actual size. This also can create flow issues, with agents lining up behind one another if densely populated, but with unrealistic amounts of free space between them. Alternatively, if only one grid cell is assigned to an agent at once, based upon whichever cell is populated as a majority, problems can occur when passing between cells. In circumstances where a section of space is densely

populated, graphical collisions may occur between agents that overlap grid cells, even though the collision detection intelligence deems them to primarily populate adjacent cells.

2.3.2.2 Proximity Detection Technique

An alternative approach to coping with agent interactions is to use traditional geometric intersection tests to determine when collisions are likely to take place. This can be achieved using bounding spheres or bounding boxes around agents, or groups of agents to break down collision testing into segments. For example, a complex model such as one representing a human can consist of many polygons. When testing for collisions against this model, it would be computationally time consuming to test each individual polygon face with that of the colliding model. On a macro scale simulation, determining which part of an agent has collided is inconsequential, rather the fact that the agent has collided, or is about to collide, is more relevant. By wrapping the model in a virtual bounding sphere or box, the complex representation is simplified to a single primitive polygon that can be easily tested against. Various forms of bounding volumes can be used to catch potential collisions before they occur at longer ranges from the agent being tested. As proximity methods are more flexible than the grid based method presented previously, better realism can be achieved more easily during collision resolution. As Foudil et al. [Foudil, 2006] notes; in real life, people will prefer to pass each other with least deviation from their path. Provided a bounding box is large enough to detect collisions at an early stage, acute adjustments to an agent's path will cause head on collisions to pass accurately, with little noticeable difference to trajectory. In contrast, grid based methods do not generally allow for such minute adjustments.

2.3.3 Other Steering Factors

Further to the steering and movement of the agents, are the sociological and psychological considerations of the crowd. Most simulations, by design, disregard the behavioural differences between individuals that you would expect in the real world. A behaviour engine will aim to address the missing 'human' components of a simulation by considering extra traits of each agent in the steering AI. This can affect factors such as the speed of the agent, or the routes he chooses. This can also be enhanced by implementing a variety of different agent animations, and associating these with specific scenarios. When certain environmental or personal variables are met, these animations can be executed, giving diversity to the scene. An industrial example of this is the program Massive Prime, which can generate behaviour for tens of thousands of agents with an overall objective. When used for rendering battle scenes for the film industry, each combatant has their own customizable responses for specific situations, some showing

fear and fleeing, while others are running towards the opposing force. Although entirely pre-rendered, these simulations implement high qualities of behavioural AI, with large crowds displaying emergent behaviours such as flocking.

As complexities of simulations increase, and with developments in processing breaking new boundaries, some simulations are being optimised to run on multiple GPUs or CPUs. For example, Reynolds [Reynolds, 2006] describes that during his research for Sony Computer Entertainment, US R&D, he is able to create very large crowds, rendered in real-time using the multi-cell architecture of the Playstation 3 CPU. Yilmaz et al. [Yilmaz, 2009] were able to create a real-time simulation tracking over one million agents by using Nvidia's CUDA libraries. These allow developers to use the powerful GPU found in most modern high end PC's for parallel processing. This creates impressive results that would be near impossible to achieve by regular means without switching from real-time to pre-rendered implementation. As similar research in this field is undertaken, the potential for the overall complexity of steering methods being produced is likely to increase allowing more lifelike and accurate representations of the real world.

3 Design and Planning of Implementation

Considerations for how the scene will be displayed and presented are discussed within this chapter, along with flowcharts of how the AI decisions will be made. Finally, criteria for measuring the success of the simulation will be laid out. These will concern factors such as realism, computational efficiency, and scalability to other scenarios.

3.1 Scene Visualization

Displaying the scene and agents to the user is essential in providing an interface for qualitative measurement of the realism. While the behaviour of the agents and their movements could be tracked and reported upon from a command line based interface, this could only be used to measure computational efficiencies of the implemented algorithms. The visualisation of the scene needs to be easily navigated and manipulated to show movements and behaviour of agents from various angles. It will also need to be designed to fit the context of the behaviour, while not being too overly complex or dense as to take away processing cycles or attention from the agents themselves.

3.1.1 Environment Context, Design, and Display

In keeping with virtual environment and crowd research being undertaken in the department of computer science at UEA, the context of the scene will be an urban city environment. This offers scope for a variety of building shapes and sizes, with the option to add areas of open space, which should give the steering methods implemented sufficient diversity for a fair test. The scene will need to be large enough to accommodate a fairly dense population for stress testing each steering method, with sections large enough to show multiple agents meeting along the same path, and resolving such encounters successfully. Graphically, the buildings will require textures to improve the aesthetics of the scene, but as this is a behaviour orientated simulation, complex building models and textures will not be necessary.

3.1.2 Camera Control

Paramount to the analysis of the simulation will be the ability to navigate the scene in real-time, and to have full control over the movement of the camera. To make the camera movement as intuitive as possible, axis rotation will be controlled using the mouse, while acceleration forward or backwards will be tied to keyboard commands. Additionally, it may be beneficial to program pre-set camera locations that can be snapped to with keyboard controls, further improving the navigation of the scene. It will be necessary to allow the

camera to leave the proximity of the scene, and not to be bound to any area of it. This will be to allow aerial views encompassing the whole scene to enable analysis of macro crowd movement. This could pose potential issues with graphical rendering, due to the amount of data being drawn in the viewpoint, and will require investigation at the time of implementation.

3.1.3 Population

The most essential part of the scene will be the agents themselves, and therefore their representation is the primary graphical focus. As discussed, agent density is more important than graphical quality, so a balance will need to be met when the optimal quantity of agents is calculated. If a low quality representation of an agent is implemented initially, it can be used to decide how many agents the scene and the AI can support. At this point, the agent's graphical representation can be scaled up progressively in quality until a sufficient rendering level is reached that is aesthetically acceptable, without being detrimental to the frame rate.

Establishing the optimal number of agents that the scene can support, (disregarding graphics as a factor), will be dependent on four main factors: the number of vertices available to visit in the steering method's matrix, the amount of free space between these points, the size of the agent avatars, and the complexity of the local collision avoidance algorithms. For example, if a scene consisted of just two sites to navigate agents between, a real-world equivalent of 20 meters apart, a population of 1000 agents would be unmanageable within very few frames. Deadlock would occur between agents competing for space, but constantly registering local collisions. Inevitably in this scenario, clipping of agent avatars would also be likely, potentially causing parts of their models to pass through one another. This is the most undesirable side effect as it shows the limits of the collision avoidance code to have been surpassed. The complexity of the AI will need to be high enough so that clipping of agents is kept to a minimum, but not too processor intensive that it causes frame loss. As with every other element of this simulation, a balance of speed to complexity will need to be met. Once the main simulation components are in place, the ideal number of agents can be established, and the behaviour or environment can be adjusted if required.

3.2 Intelligence and Behaviour

The method used for high level navigation of the scene will be Dijkstra's shortest path algorithm. This will navigate agents across two sets of paths generated by a visibility graph, and the Voronoi algorithm. The difference between the two arrays of path vertices

will be inconsequential to Dijkstra's as both will produce a set of points for the algorithm to compute paths, at runtime. No additional parameters should be required for either method. When implemented, the main difference between the methods will be the way in which they direct the flow of the agents. The visibility graph will be built on the surrounding points of structures, and will therefore manifest as showing agents moving close to the buildings, and crossing open areas to corners of structures. In stark contrast, Voronoi will funnel the agents to the furthest possible points from the structures, potentially using less path vertices, but also increasing the density of agents on a given path.

The visibility method for calculating paths, will establish which points have a line of sight to other points in the scene. This will be calculated by drawing straight lines from one point to every other point in the scene sequentially, and checking whether these are intersected by structural lines of buildings. The results of this will be stored in a matrix and then referenced by Dijkstra's shortest path algorithm. Implementing this calculation so that it can be completed before the scene begins in real-time, will prevent the need to evaluate these details per frame. This should greatly reduce the demand on the computer hardware. Ideally, the method for calculating this should be dynamic and reusable so that new points can be added and included into the scene without changes to the algorithm's structure. In terms of pre-run time calculation, the Voronoi method will be similar and should allow the heavy weight computing to be completed before the graphics of the scene begins. Given the same selection of structure points, the Voronoi method will return a set of resultant paths that will include passages through building centres. This will require a second round of processing, testing all path lines against building lines to remove any failures, in the same way as planned for the Visibility method. Because of this, it is expected that this method will be the longest to compute.

When an agent requires a new path, Dijkstra's shortest path algorithm will be used to query the path matrix (calculated by either of the steering methods), and then the agent will set off towards the first location in that path. By storing the results of this calculation as a sequence of path nodes, it will not be necessary to call on the Dijkstra's algorithm again until the agent has completed the entire path. In theory, calling the algorithm at every node in the path would render the same path result (minus previous destinations), just with a far greater use of processing power.

Once an agent is travelling on a path calculated by Dijkstra's algorithm, collisions will be handled by the proximity or grid methods of collision resolution. These will only detect and resolve collisions with other agents as the high level steering methods will have ensured that an agent should not choose a path that would cause a collision with a building. By

removing the need to check against structures in the scene, the requirement for complex line intersection code is eliminated, drastically cutting down the calculations required per frame.

The proximity detection method will require the agent to test whether any other agents are within their ‘personal space’, and if so act upon this. The size of this personal area will need to be established through testing, to determine ideal parameters. If it is too small, agent models may overlap one another before detection takes place, which is detrimental to visual realism. An overly large detection space may cause the agents to space out far too widely and collide with buildings, or cause issues with larger density crowds. Resolution will be conducted by the agent that detected the collision, moving either to the left or right of the detected obstacle, depending on their incoming angle. Theoretically, if this movement is enough to clear the other offending agent, then that agent will not need to conduct any collision resolution at all, cutting down on processing. At worse, both agents will determine that a collision is imminent and shear either side of each other, as shown in Figure 9. Once the collision is resolved, an agent will need to be placed back onto its original path, only further deviating if another collision is detected.

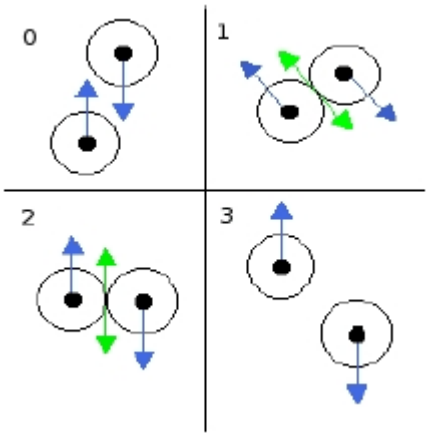

Figure 9. The stages of collision avoidance, showing agents shearing past one another. Adapted from [Champagne, 2005].

The grid method of collision detection will test on a far more short sighted range, checking whether the grid that the agent wishes to travel to is free to enter. This grid will be a virtual representation of the scene that only the AI will interact with, therefore some pre-processing will be required to translate visual elements into the grid, for example; grid locations that are populated by buildings. This can be calculated in a pre-processing step, and will prepare the grid with static obstacles that cannot be entered. As each agent moves around, their grid location will update, setting their currently occupied grid cell as an impassable location. Other agents will check this grid map as they move between grid

locations, to ensure they can pass freely into their next cell. This has a benefit over the proximity detection method, in that the agents will be aware of building locations, although as previously discussed, they should not pass near buildings during their route. For local collision detection, agents need only check adjacent squares and not the location of other agents as in the proximity method. This should be a much more efficient method.

Flowcharts covering the AI decision trees for Dijkstra's shortest path algorithm, and the collision avoidance methods, are covered in the following section. They will be used as a guide for programming the AI functions during the implementation, and represent the preferred order that decisions should be made in.

3.2.1 Steering Flowchart

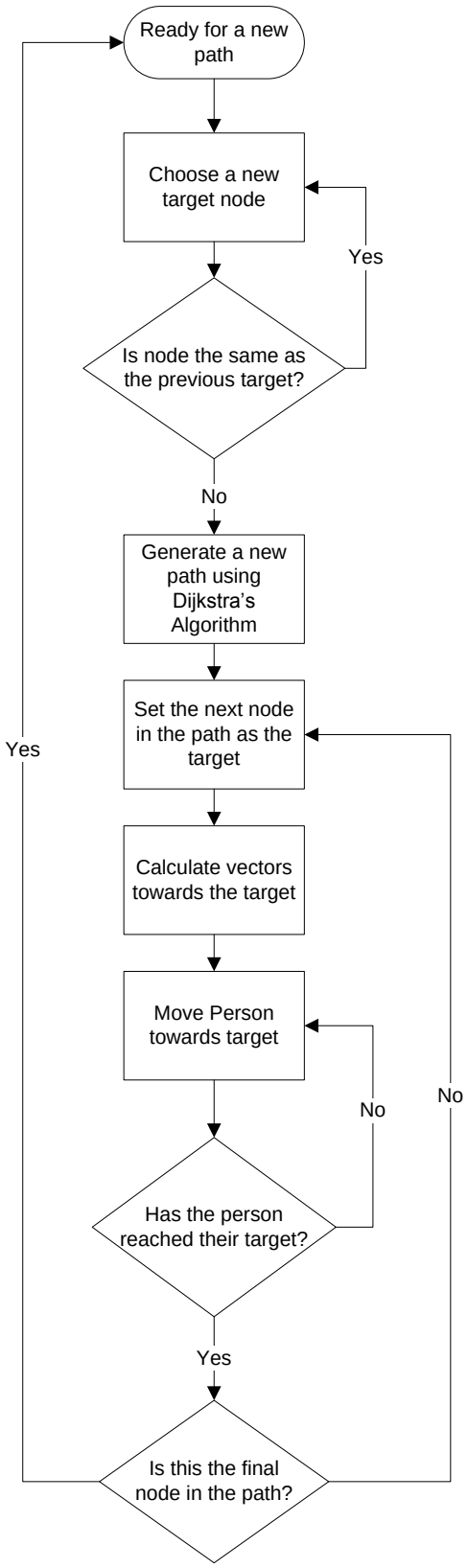

Figure 10. Flowchart of high level steering AI.

3.2.2 Avoidance Flowcharts

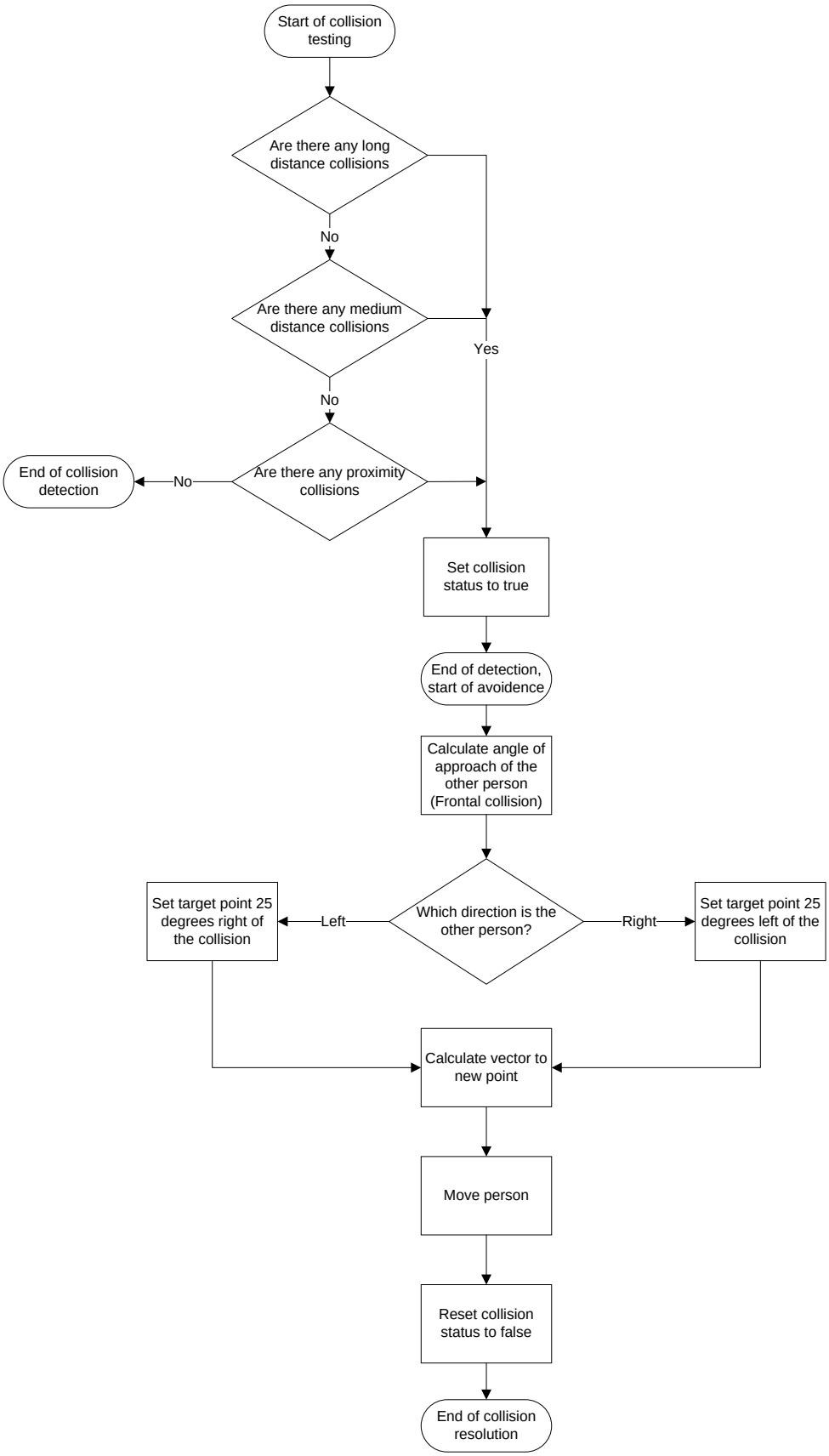

Figure 11. Flowchart of the Proximity collision detection and resolution method.

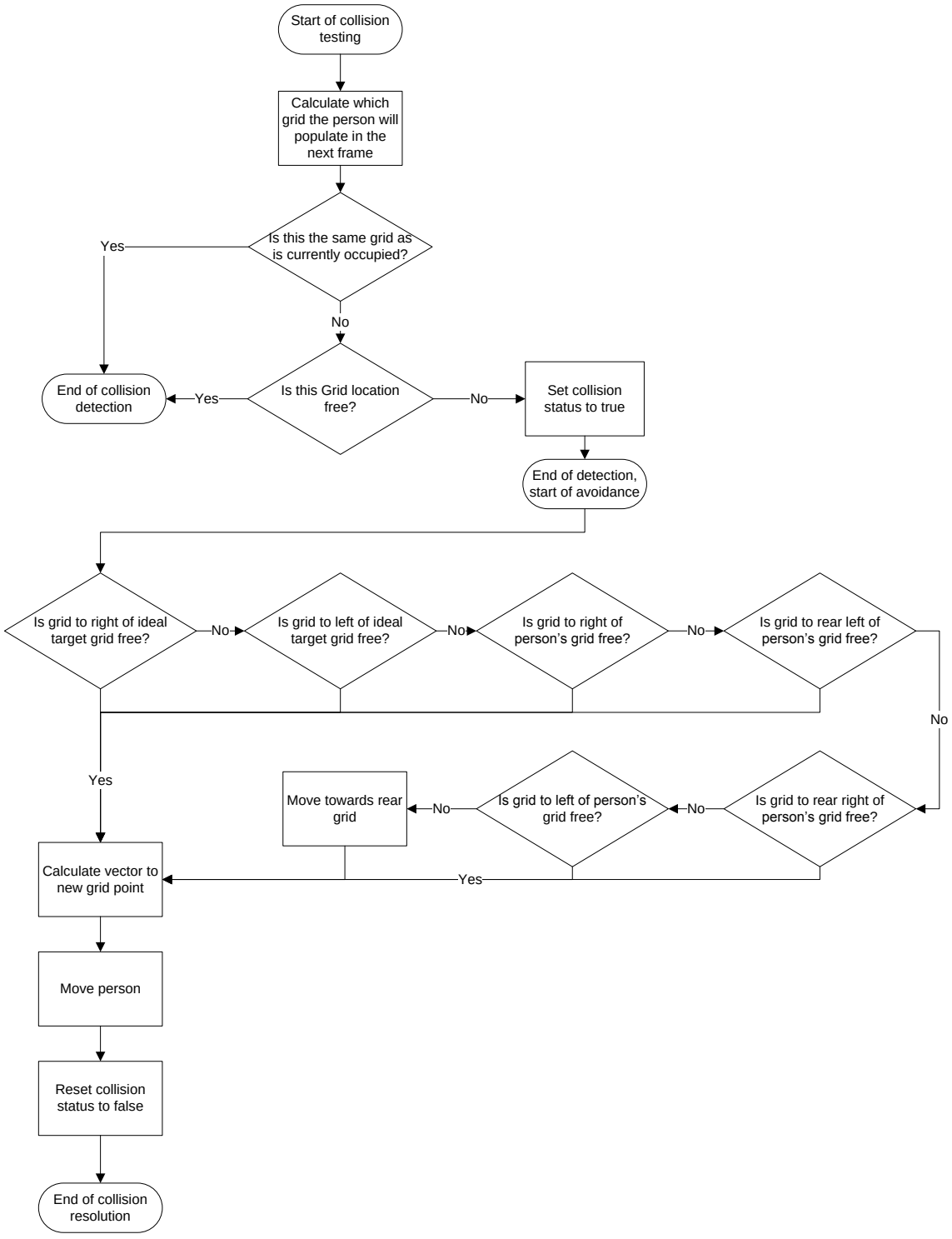

Figure 12. Flowchart of the grid collision detection and resolution method.

3.3 Measurement of Results

As described in the objectives of this thesis, the aim is to measure which combination of both high and low level steering is superior based on a number of quantitative and qualitative factors. To achieve this, both computer and human methods will be used to measure the success of the implementation, and a conclusion will be drawn from these findings. As stated in the objectives, each simulation test will be run for an equal amount of time, with the same number of agents, although this number will need to be established post-implementation, and is dependent on how many agents the engine can effectively render and control. The test machine will consist of an AMD Phenom II 3.34Ghz 6 core processor, with 6Gb of RAM, and a GeForce GT250, running in Windows 7 Ultimate 64Bit.

Quantitative measurement will encompass the computational side of the result analysis. This will consist of measuring each method (and combination of methods), for speed of pre-processing, overall memory requirements, and runtime frames per second. Each test will be repeated 10 times on a system running only essential background processes and programs, and the mean of each test will be established from these results. Processing speed will be measured by placing timers within the code at the beginning and end of the pre-processing classes, and outputting this to a file or the console for recording. Memory measurement will be achieved via the Windows Task Manager which can be configured to give a comprehensive display of paged and unpaged memory use per program, including total memory consumption. Finally, the FPS at runtime will be measured using a free-ware application called Fraps which is commonly used within the computer graphics community.

Qualitative measurement will require human testers to provide feedback on which methods they think look the most accurate based purely on visual preference of behaviour. For this purpose it will be important to ensure that the range of individuals selected are from varying backgrounds of technical expertise, as to only test Computer Science scholars, for example, could give a potentially construed set of results and opinions, as their insight into the subject could influence their decisions or opinions. Each combination of methods will be presented in a video captured from a real-time rendering of the scene. The most efficient way to record the data from these demonstrations will be to request ratings of each method combination, so as to create a set of results that can be statistically analysed.

3.4 Code Practices and Methodologies

Throughout the planning and analysis sections of this thesis, it has been established and stated that efficiency is one of the highest factors of a simulation's success, and this can only be achieved through well designed code. To achieve this, each section of the simulation will be separated into its own class containing public methods that other classes can interact with. Using this design model will allow high reusability of code, for example, a Crowd class could hold an array of Person's (another class), allowing for that Person and its attributes to only be described once in the code, but reused multiple times. Making each section modular in this way will allow the different methods of steering and collision detection to be switched between while still using the same core components such as lighting and rendering classes. This project will be coded using Visual Studio 2010 Ultimate, in C++ with OpenGL, using Team Foundation Server 2010 for source control.

4 Framework Creation

The implementation of creating the scene is set out in this chapter. Processes for creating buildings, landscape and a camera system to navigate them are described, along with the lighting methods used. This chapter will also cover the graphical implementation of the agents, describing how best to represent them within the scene depending on the density of the crowd being displayed.

4.1 Buildings

The primary objectives of the structures in the scene are to give the agents context for their movements, and act as obstacles to steer around. As previously discussed, the simulation theme will be an urban city environment, consisting of various buildings of different sizes and shapes, so as to create diverse paths for the agents to follow. The complexity of the scene will have a direct impact on both the visibility and Voronoi steering methods, as both algorithms will be run against the vertices of the buildings.

4.1.1 Building Location and Design

To ensure that the agents can move relatively freely between locations, all buildings should be spaced so that at least four agents could pass through a space in a line formation, at a minimum. Although none of the proposed steering methods would create any direct situation where agents would be deployed in this manner, it will allow for avoidance to take place in densely populated situations and theoretically prevent bottle necks from occurring. As described, if the scene becomes overpopulated, this situation will no doubt become unavoidable anyway, but the scene must be designed with the 'model' situation in mind. Buildings consisting of curved walls will not be used as they pose two problems for the program. Foremost, they are vastly more complex to render, consisting of many more vertices and therefore polygons. As there is a finite amount of per second rendering ability available on the test machine, it would seem more logical to allocate as much of this as possible to the rendering of agents, and to maintain an optimal rendering quality with an acceptable amount of FPS. Secondly, as both path creation algorithms calculate results based on the structure vertices, a curved wall would create a dense number of path points in a small area of space, leading to problems with agent guidance. In an example where a curved wall consists of 10 segments, that wall section would contain 11 vertices. The complexity of computing a Visibility Matrix increases by $O(n^2)$, and is therefore around 100 times slower to compute. Additionally, Dijkstra's algorithm will interpret these as 11 different destination locations, and could potentially (worst case) send an agent on various different routes between these points, seeing them all as individual potential targets. The solution to this is to approximate the group of points

into a few or one point, however, this would either need to be done via a separate algorithm (which is unlikely to work in all scenarios), or manually. Scaled up to a very large scenario, manual setting of points would soon become very impractical. In simulations representing real life scenes, building artists will take such factors into account, assuming the scene is manually drawn. Procedurally drawn scenes may undergo post processing by algorithms to identify and simplify curved edges.

To offer variation in building shape, the scene will contain two irregular buildings, one of which is very large consisting of many points. This will offer the scene variety that is desired and complex routes for the agents to navigate. The remainder of the scene will consist of six buildings of varying sizes, with four vertices each, laid equally apart. The border of the scene will be impassable, containing all of the agents within it, so a perimeter wall will be added to the top and bottom of the scene. The side boundaries will not be drawn for the ease of navigation and visualization by the viewer; however this will still be impassable by agents as if a virtual wall was in place.

Figure 13 shows the design for the planned scene in 2D top down format. It contains 46 vertices (including the bounding perimeter corners), and offers a range of small and large open spaces. This should allow plenty of opportunity to see a full range of behaviours displayed from the planned steering and collision avoidance implementations.

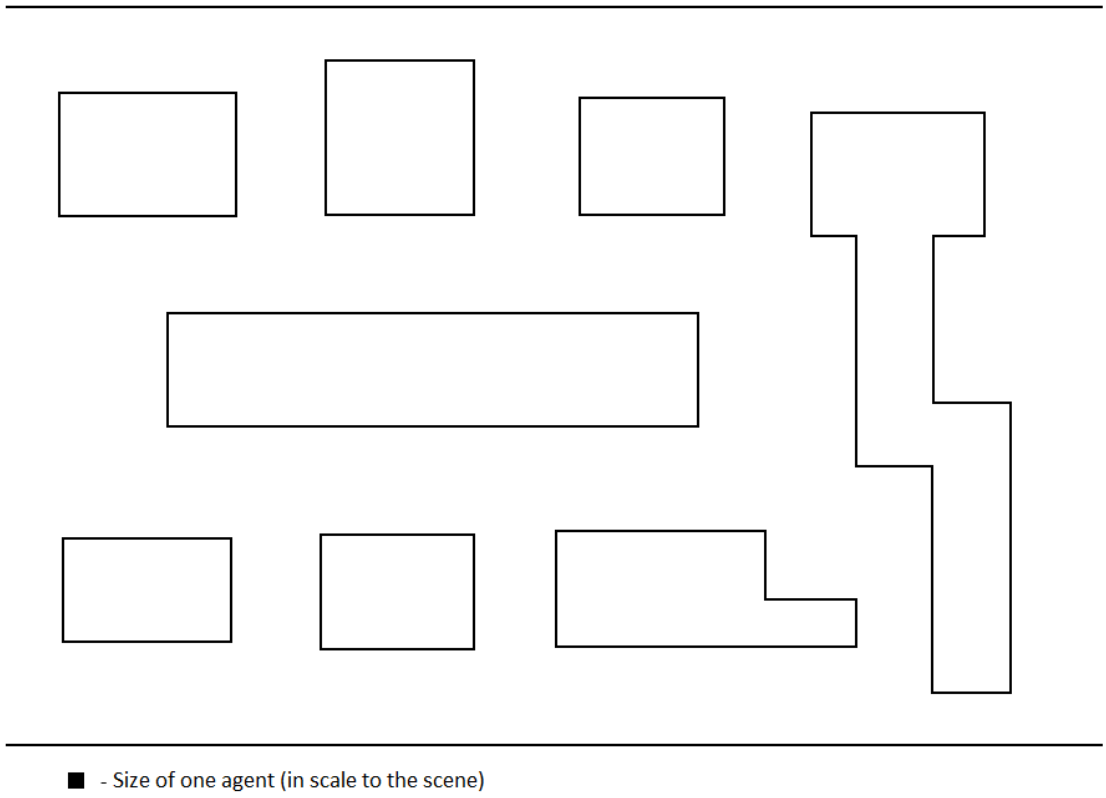

Figure 13. A top down view of the planned layout of the scene for use in this simulation.

4.1.2 Scalable Wall Creation Algorithm

With the building layout finalised and building vertex coordinates stored in an array, the scene required the walls to be constructed. To draw the scene as shown in Figure 13, as simply as possible, 45 single faced polygon quads would be needed, including one for the floor. This presents an issue in that the two shading models in OpenGL (Smooth and Flat), show particular weakness across large single polygon faces. The Flat shading model uses the normal calculated light to shade the entire polygon face. This would be the equivalent of lighting one end of a long wall, but seeing the entire length of it brighten up equally. Smooth shading illuminates the face based on the light level at each of a polygon's vertices, meaning that a large single face lit centrally will evenly lit across the face. Neither of these lighting models offer realistic results used on a simple building structure, so to achieve acceptable visual appearance, each wall would need to be created from a multitude of polygons. Per pixel lighting utilizing Phong Illumination is another method that could be used, and would only require walls consisting of a single polygon. This uses GLSL, and was deemed too complex for this implementation due to the processing overheads that would be incurred. As polygon normal calculations are best run against triangular polygons, it was decided to create a wall constructed of matched triangle pairs, creating a square structure. To do this efficiently and accurately, an algorithm was created (Algorithm 1) that could be reused for each wall structure in the scene.

```
for (float k = 0; k < xsize; k += xseg){
    for (float l = 0; l < zsize; l += zseg)
    {

    glBegin(GL_TRIANGLES);
        vec.vec[X] = (xcord); vec.vec[Y]  = (ycord); vec.vec[Z]  = (zcord);
        vec1.vec[X] = (xcord); vec1.vec[Y] = (ycord); vec1.vec[Z] = (zcord+zseg);
        vec2.vec[X] = (xcord+xseg); vec2.vec[Y] = (ycord); vec2.vec[Z] = (zcord);
        objNormal->ComputeNormal(&vec3, &vec, &vec1, &vec2);

        glNormal3f(vec3.vec[X],vec3.vec[Y],vec3.vec[Z]);
        glTexCoord2f(0.0f,0.0f);
        glVertex3f(xcord, ycord, zcord);
        glTexCoord2f(0.0f,1.0f);
        glVertex3f(xcord, ycord, (zcord+zseg));
        glTexCoord2f(1.0f,0.0f);
        glVertex3f((xcord+xseg), ycord, zcord);
    glEnd();

    glBegin(GL_TRIANGLES);
        vec.vec[X]  = (xcord+xseg); vec.vec[Y]  = (ycord); vec.vec[Z]  = (zcord+zseg);
        vec1.vec[X] = (xcord); vec1.vec[Y] = (ycord); vec1.vec[Z] = (zcord+zseg);
        vec2.vec[X] = (xcord+xseg); vec2.vec[Y] = (ycord); vec2.vec[Z] = (zcord);
        objNormal->ComputeNormal(&vec3, &vec, &vec1, &vec2);

        glNormal3f(vec3.vec[X],vec3.vec[Y],vec3.vec[Z]);
        glTexCoord2f(1.0f,1.0f);
        glVertex3f((xcord+xseg), ycord, (zcord+zseg));
        glTexCoord2f(0.0f,1.0f);
        glVertex3f(xcord, ycord, (zcord+zseg));
        glTexCoord2f(1.0f,0.0f);
        glVertex3f((xcord+xseg), ycord, zcord);
    glEnd();

    zcord += zseg;

    }
    xcord += xseg;
    zcord = 0.0f; }
```

Algorithm 1. The wall creation algorithm for creating shaded walls consisting of multiple segments.

By accepting the parameters of length, height, the number of horizontal segments, and the number of vertical segments, the algorithm could create a wall of any desired size, and subdivide it equally, as specified. Passing in values of 1 for segmentation would create a wall consisting of one segment, constructed with two triangles. Briefly summarised, the function takes a distance (*Xsize*), and divides that by the quantity of segments (*Xsegments*), to give a segment size (*Xseg*). This is repeated for the height parameter (Zsize), resulting in the total size for a segment (*Xseg* x *Zseg*). As shown in the code listed here, a loop is then entered in which a segment (pairing of two triangles) is drawn, and the normals of both triangles computed (by passing the triangle parameters into another function). The loop ends and is restarted with *Zseg* added to the drawing coordinates, which begins the drawing of the next segment layer, continuing until the desired wall height is reached (*Zsize*). This is repeated per row (*Xseg*), while the columns are drawn bottom to top (*Zseg* distance apart), hence the nested loop required to perform

this operation. Also laced into this code are texture bindings for texturing the structures at a later stage of development.

Overall this produces a far superior wall model compared with using a single faced polygon. By setting the desired x and y segment amount high enough, a smooth wall could even be drawn using the Flat shading model. This does however require a large quantity of segments, and an equally effective result was achieved by using far less segments with Smooth shading enabled. Testing showed that the number of segments required per wall, was the length or height, divided by 10. For example, a wall that was 1700*x* by 300*z* would be constructed from 170 x 30 segments. This gave acceptable levels of shading across the wall's face while maintaining as low a polygon count as possible. It was also still considered computationally cheaper than implementing per pixel lighting. Using more densely packed segments offered little extra improvement in quality while directly increasing the computation required, however, less segments than this began to clearly show visible borders for each segment. The amount of segments used for the floor was reduced slightly as the viewing proximity will be such that the level of shading will not be so noticeable. A comparison of using Smooth shading against Flat shading, combined with a basic test light source, can be seen in Figure 14. Each wall was created in this fashion, and then moved into place using translation and rotation, relative to the centre of the scene.

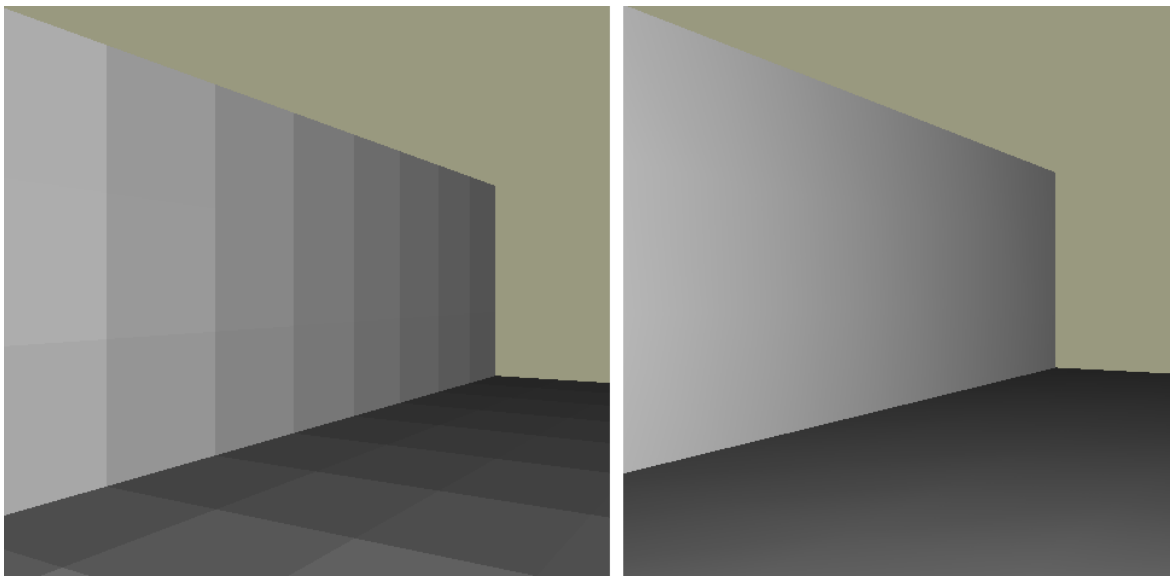

Figure 14. A comparison of OpenGL Flat and Smooth shading models. Mach Bands can be clearly seen on the Flat shading picture (left).

With all structures in place, the next stage of development was to add textures to the walls and floor. While not necessary in the context of the aims for the project, it should assist the qualitative testing as it will improve the visual appearance of the scene. Due to the

way in which the walls are constructed, a texture would need to be mapped per segment, and repeated across the surface. Further to this, each triangle in a segment requires one half of a texture to be mapped to it, meaning that the total times the texture is mapped is the number of segments multiplied by two. To reduce the number of texture binding calls, which can slow down frame processing, all structures were drawn sequentially after having the wall texture selected. Drawing buildings between agents drawing calls, for example, would require a greater frequency of texture binding. Finally, a suitable wall texture was chosen and applied.

4.1.3 Roofing the Structures

To add roof tops to the scene, a derivative of the wall creation algorithm was used that would create one quarter of a peaked roof. In the version used for wall creation, the algorithm composed of two right angle triangles, attached by their hypotenuse, to create a square. For roof creation, the modified algorithm was altered to draw both triangles in reverse of each other to create an isosceles triangle. The parameters fed into the function were altered to supply the desired width of the roof section, and the depth of the centre point. The height of the roof was fixed so that the buildings looked uniform across the scene, as shown in Figure 15. Because the roofing algorithm was a derivative of the wall algorithm, it was already set up to calculate lighting normals and attach textures.

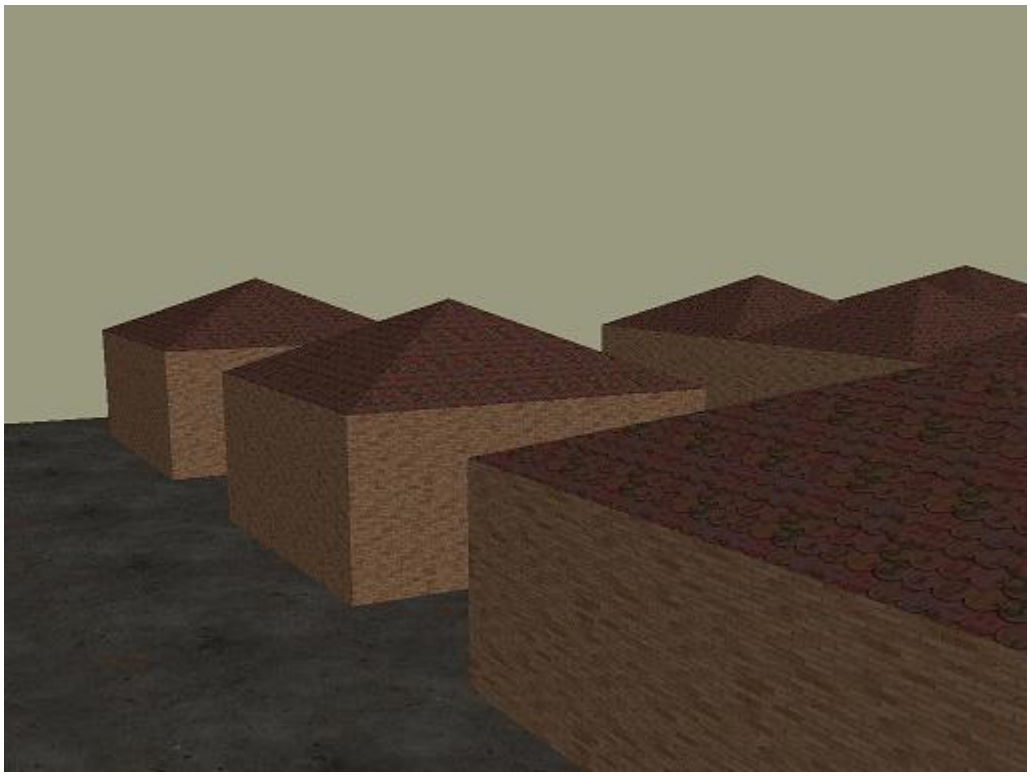

Figure 15. The completed scene displaying buildings with roof tops and textures.

The completed algorithm allowed a roof to be added at the same time as a corresponding wall section, providing the width of the wall as the roof width. The roof length (distance into the centre of the roof) was taken by using the centre point of the target building. This created well shaped symmetrical roofs for square structures.

To improve run time efficiency of rendering the environment, the code for the walls and the roof tops were added to a Display List. This allowed for all of the scene's vertex and pixel data to be cached during program initialisation, and placed into a compiled state in memory. Each time a frame is redrawn to the screen, OpenGL can call the Display List for the buildings, without needing to recalculate lighting normals, or texture bindings. This allows far more computational power to be freed for drawing the agents, which are after all, the main focus in the scene. The only disadvantage in the use of Display Lists is that the geometry data within the List cannot be manipulated during run time, having been compiled during the initialisation of the application. Due to the buildings being part of a fixed scene, this is a low risk issue, as the buildings are neither animated nor changed for the duration of the simulation.

4.2 Camera System

To enable complete realisation of the scene, the viewer must be able to navigate it intuitively, allowing for any given angle to be attained. To achieve this best, a First Person style camera was used. This style of camera uses the position of the viewpoint as the centre axis, meaning that any movements to the scene are calculated relative to the viewer. This is the closest representation of what a human would see if they were in the scene, with camera manipulation showing scene translation that would be expected in a real world equivalent. It is common for this style of camera to use the mouse to represent head movement (scene rotation), and keyboard bindings to control position movement around the scene.

4.2.1 Quaternion Camera Implementation

For the purposes of this simulation, it was decided that Quaternion's could offer the best solution to create the desired camera system. OpenGL does not have a camera system that can be manipulated to move around a scene. Instead, the scene perspective is changed by moving the scene around the viewpoint. To use Quaternion's for this, the scene geometry needs to be multiplied by the matrix created from the Quaternion, before being rendered per frame. By placing a call to the Quaternion code in the OpenGL Display Loop, before any other drawing takes place, all structures and objects in the scene are translated by the Quaternion matrix, relative to one another.

An Implementation of a Quaternion camera system was obtained [OpenGL, 2011] for integration into the simulation framework. After various changes to bring it in line with the project's code methodologies, it was apparent that the camera rotation control was not compatible and would need rewriting. The Quaternion class translates mouse movement into camera rotation by analysing the movement of a mouse cursor within the window, per frame. This was converted from a DirectX implementation into an OpenGL equivalent. For movement around the scene, the *W* and *S* keys were assigned to forward and reverse motion, completing the camera implementation. As specified in the design, and to speed up scene navigation, some pre-set fixed camera locations were needed that could be reached with keyboard commands. To establish these coordinates, the chosen locations were navigated to manually, and their coordinates recorded.

4.3 Lighting

As displayed during the building implementation, all structures and scene elements thus far were designed to respond to light sources. This helps the viewer grasp a level of scene depth, making the scene seem less 2D when viewed. To best achieve this, a level of low ambient light was applied to all objects, bringing their material brightness up to an acceptable minimum level. A single light source was then added to the location of the viewpoint, meaning that as the scene was navigated, the light source would move with the view. This would ensure that proper lighting was present from every viewpoint, removing the risk of unlit areas creating poor visibility, and removing the need to implement multiple lights in the scene.

4.4 Agent Rendering

4.4.1 Spherical Representation

To assist with the implementation and debugging of the AI, a temporary representation of the agents was required. Since the focus of the simulation is on the intelligence of the agents, the final character models were chosen after the implementation of the steering methods was complete; so their impact upon the run time computational power of the simulation was measurable. Applying models in this way ensured the AI was unrestricted and allowed for the best possible agent models to be selected.

The temporary representation of the agents was chosen to be a standard Sphere from the GLUT library. This is an inbuilt primitive that has precompiled normals, and can be

invoked with a single line of code. Their diameter was set to be equal to the intended human model size, in proportion to the implemented structures. With low graphical overhead, the inbuilt sphere proved to be a perfect three dimensional representation for designing and adjusting the behaviour. Due to this ease of rendering, the total quantity of spheres that could be drawn at once, was reaching, or surpassing the upper limits of what the AI could cope with, in terms of agent density. Figure 16 shows a screen shot from an early design stage, in which the density of the agents is at a level where individual paths are becoming too tightly populated, and collision resolution methods would be under extreme pressure to resolve all collisions while maintaining an acceptable level of performance and a smooth progression of agents along their paths. The sphere model is essential in this example, as this would not be realisable with high polygon, full rendered human models on the test hardware available.

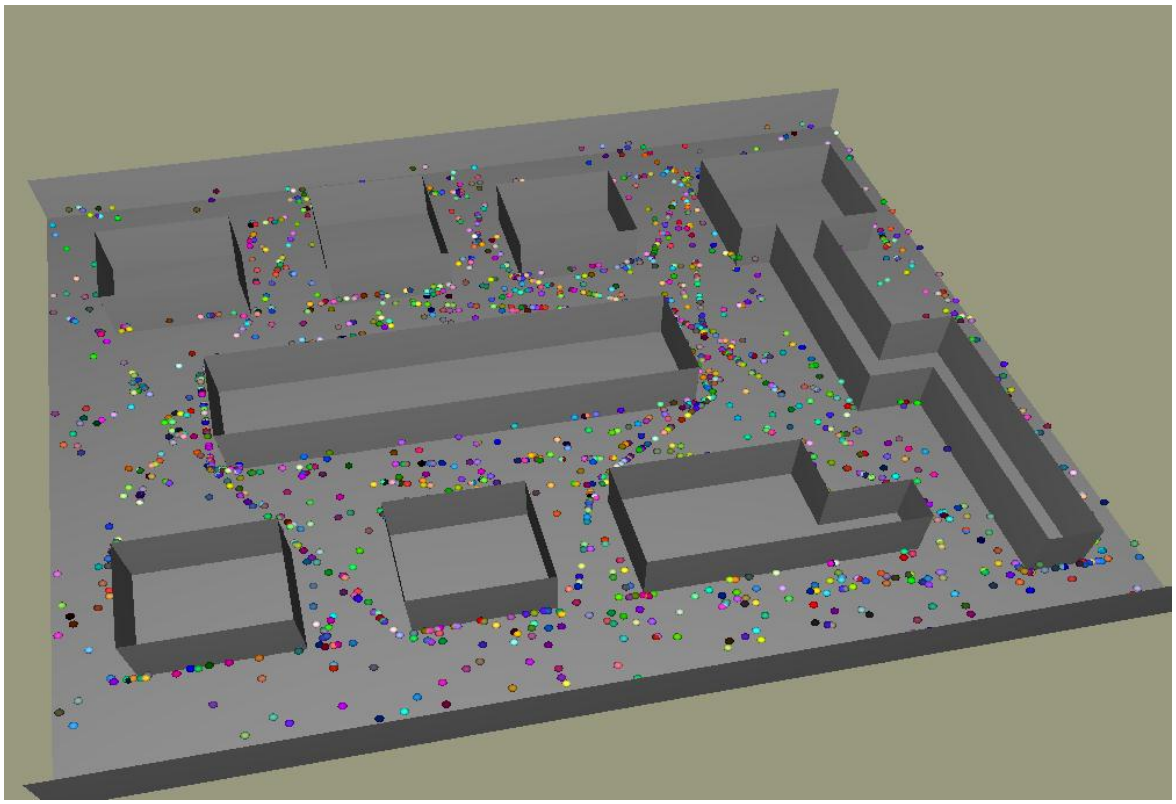

Figure 16. A basic building scene hosts 2000 agents moving along paths generated by high level steering. Without hardware greatly in excess of that used in this project, this scene would be difficult to render with full scale realistic human models, while still running in real-time.

4.4.2 Human Representation

As discussed, to gain full appreciation of the scene during qualitative testing, realistic human models needed to be integrated into the scene. Models were kindly donated by the Computer Science Department of UEA. These had been purchased for similar departmental projects from AXYZ-Design, and were supplied in the form of component

files representing; the three dimensional model, the character texture (clothing, skin), and the animation for movement. These required assimilating into C++ for use in the scene, using character loading code. This was kindly donated by Ruben Galvao of UEA, and required some integrating into the existing framework. In total there are 48 unique people that can be rendered by the loader code, all of which can have multiple instances in the scene at once, the limit therefore, is on the graphical hardware of the test system. Figure 17 shows the layout of the simulation using human models, from a similar view to that seen in Figure 16.

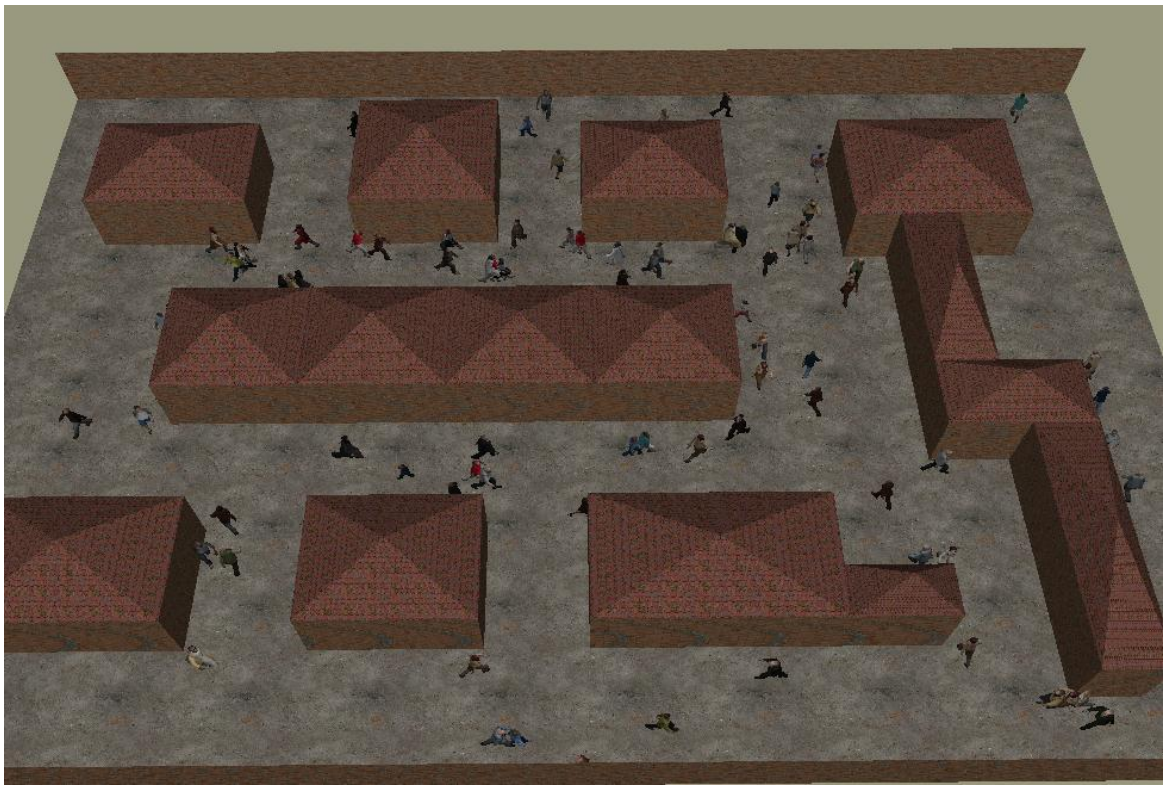

Figure 17. A bird's eye view of the scene shows 96 human agents following their paths. Each model is repeated twice as there are 48 unique characters.

The computational requirements of the graphics hardware to render the human models is roughly 40 times that of rendering the spheres used during testing (comparing polygon triangle proportions of 225 for spheres, to ~9000 for the human models). Therefore it can be deduced that on the test machine used in this project, the density of agents in the scene is unlikely to reach a critical level, when using the human model. A critical or overpopulated density would be akin to the situation described previously in which the AI is unable to handle the quantity of agents in the scene.

To counteract a drop in FPS when adding more agents to the scene, other visual frame dependent factors can be increased. To better explain this, the effect of low FPS will be described in the context of this scene. The main implications are: slow camera movement

as the changes in mouse position are being checked and acted upon much less; slow avatar movement, as their position is being updated less per real-time second, slow animation is most noticeable due to the animation speed not being frame rate independent. This can be compensated if all of these features are adjusted, such as increasing the animation speed and the amount of camera movement per second which would make the drop in FPS less noticeable. The potential issue in doing this is the drop in frequency of the AI decisions. Fewer frames are being processed but the distance travelled by agents has remained the same, therefore collisions will be detected much later then at a higher frame rate, and could result in being resolved too late, or not at all in extreme cases. With these factors considered, spherical models should replace human agents for crowd densities that drop the frame rate below 25 FPS, provided this is preferable over reducing the number of agents. This will ensure that the behaviour of the agents is kept as the primary focus of the simulation, with graphics as a secondary computational priority.

5 AI Design

This final implementation chapter will cover the most important section of the thesis, describing the creation of the agent AI by utilizing steering algorithms set out in chapter 2. The chapter is split into two halves; one covering high level steering and the second describing local, low level techniques.

5.1 Steering and Macro Control

The first stage of the AI development was to get the agents moving around the scene from a location to a target. Dijkstra's shortest path algorithm was to be used for this path finding, but the points themselves needed to be generated first. In a scene of this size, points could be selected manually and entered into an array for the algorithm to work against; however the aims of this thesis stipulate that the methods presented should be scalable for use on different and potentially larger scenes, with as little manual intervention as possible. The visibility graph is the first implemented method, and is generated from the vertices of each of the building's corners, meaning that paths will run tightly and efficiently around buildings. Applying this to any scene requires a list of building corner co-ordinates, making this method easily scaled up to other implementations, provided the building vertices can be easily fed into the algorithm. Voronoi graphs generate routes that represent the furthest point from an impassable area. These sites are represented by single points in the middle of the obstacle to be processed. Due to this, different locations would need to be generated for use by the Voronoi graph algorithm, as building corner vertices were not appropriate. If the algorithm was run against these vertices, paths would be generated between building corner points and therefore through the centre of buildings. Using all of a building's co-ordinates to calculate the middle point for a structure, supplies the Voronoi algorithm a central location to avoid, while still providing an easily adaptable system should more buildings be added to the scenario.

5.1.1 Visibility Graph

The building edge visibility graph is required to supply Dijkstra's shortest path algorithm with an array of locations that can be navigated to by agents. Each point should be tested for its ability to reach another point, unhindered by structural obstacles, with the result of this recorded within the array. Every set of points that proves viable has the distance between them recorded, so that Dijkstra's can determine the appropriate path to take to a target, when multiple possibilities present themselves. These results are entered into a Visibility Matrix which is a two dimensional array that represents every point on the scene, and its relationship with the other points. When the array size is considered in terms of *x* and *y*, the distance of a point can be obtained by returning the data held at *matrix[x][y]* or

matrix[y][x]. If a path is impassable, an 'infinite' distance value will be returned, therefore causing Dijkstra's algorithm to select another target.

Implementation of this method began with the creation of the algorithm to process path points. As described previously, the points used were those of the building corners, meaning that they could be read from the existing array of structure coordinates. The algorithm needs to assess each possible path for structural collisions, looping through one starting point at a time. It begins by getting the first point, *a*, and drawing a line between this and the next point in the array, *b*. The second program loop is then entered into which processes each structure wall in turn, and checks for an intersection between the wall line and the proposed path line *a-b*. Using the equation of each line ($y = mx + c$) for intersection calculation means that each line is drawn infinitely, meaning any collisions detected must then be checked to ensure that the intersection is between the two path points. Once the test against a single wall line is complete, *a-b* is tested against the next structure line, until a genuine collision is found or the array reaches an end, and no building lines remain. If an intersection is detected, the test is halted and the line *a-b* is marked as impassable in the Visibility Matrix. Alternatively, if the test reaches the end successfully, the distance between the points *a,* and *b,* is recorded in the matrix, signifying a valid route. The algorithm then moves onto the line *a-c*, and restarts the test against all structure walls. This sequence continues until the test for the last path *a-n* (*n* representing the final path point), is complete, meaning that all possible destinations from the point *a* have been calculated. The entire process is repeated for the next point in the path array, until all paths from point to point have been considered.

With the algorithm complete, the paths were rendered as two dimensional lines onto the scene for visual error checking. Initially it was clear that there were three problems that needed addressing. Firstly, due to the path calculation using the corners of buildings for navigation points, many paths ran exactly along the walls of structures. This would cause graphical collision (clipping), as agents are either to be represented by spheres or human models, neither of which are equal to one pixel in width. For example, a sphere moving along one such path would display as half-in, half-out of the building, with the wall dividing the centre of the sphere, and the centre representing the position of the sphere as set by the path finding algorithm. To solve this issue, all path points were increased in a direction away from the building, by the equivalent of one rendered person's width. The second issue was a mathematical one, and was due to an oversight in the calculation of line intersections. Where lines between sets of path points could be drawn between the corners of structures, the Visibility Matrix had recorded a positive path result, despite the fact that the route passed through a structure. This had occurred because the visibility

testing algorithm had determined that although the lines intersected, the intersection did not occur within the building line. The line did intersect the point itself, although this was not counted. An adjustment was made to the algorithm to account for this. Lastly, the remaining issue was regarding empty space in the scene. Along edges of the scene, where no buildings were present, much of the space was unused as paths would not pass through these areas. In a scene of a greater size with many more buildings, or buildings that were spread out more, this would not present such an issue, as the focal point would be the denser, central sections of the scene. In this scene, due to its size, it was decided to manually add fourteen extra locations, around the edges of the map. This would space out the agents allowing larger amounts to be rendered while avoiding bottlenecks caused by density issues. Due to the dynamic design of the Visibility Matrix algorithm, these points were easily assimilated into the calculation, resulting in a final matrix of 60 points.

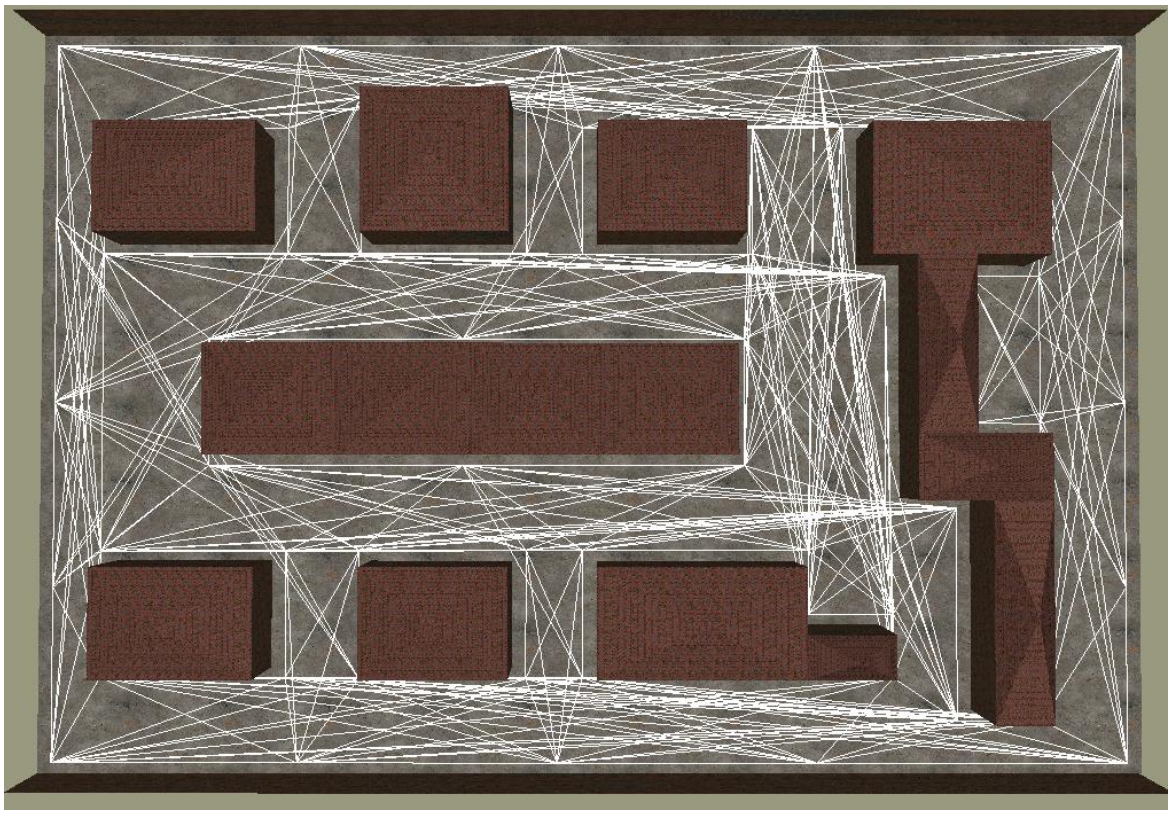

Figure 18. Lines rendered between points in the Visibility Matrix show all possible path routes that can be chosen by an agent when moving from goal to goal.

To calculate the matrix, the algorithm must test paths from all 60 points to one another, against every building wall in the scene, of which there are 32. This gives a maximum possible number of 113280 line intersection tests, although many tests are skipped if an intersection has already been detected for a line. The calculation of the matrix is completed in the pre-processor section of the simulation, before the rendering of graphics

begins, and need only be run once. This will prevent any detriment to the run time quality of the scene, and allows for more points to be added without consequence.

5.1.2 Voronoi Graph

The second path generation method to be implemented is the Voronoi graph. There are three main algorithms commonly used for generating Voronoi graphs; Bowyer–Watson algorithm, Fortune's algorithm, and Lloyd's algorithm. Bowyer–Watson is used for generating a Voronoi in multidimensional space by adding each new point sequentially, which modifies the graph accordingly. Lloyd's is a multi pass algorithm that reprocesses its output to adjust the locations of the sites, as to centralise them in the initially generated Voronoi segments. Fortune's is a plane sweep algorithm, meaning that it assesses points as they are encountered, by sweeping a line across a two dimensional Euclidean space, conventionally from left to right, although this can be set to process from any direction. Fortune's is the algorithm chosen for this thesis as it offers the most efficient and simplistic method to reach the required output graph. An implementation of Fortune's was sourced from [Fortune, 2011], and adjusted as required to work within the existing codebase. Before it could be run, sites for it to process needed to be selected. As discussed previously, these would be points representing the centre of each building. Fortune's algorithm would work upon these and create surrounding segments, the boundaries of which would produce the paths for the agents to follow. In a similar fashion to testing of the Visibility Matrix, the processed Voronoi graph was rendered onto the scene as white lines to display the output paths. The first display of the graph showed that using one point in the centre of the large rightmost building, and the large middle building, would not be sufficient as the Voronoi segments surrounding them were not large enough to encompass the structures. To solve this, three points were used for the right, and two for the centre building, evenly spaced throughout. As Figure 19 shows, this caused an inevitable problem, with path lines moving through the building. By further processing this output using the building wall intersection testing code that was used in the Visibility Matrix generation, these lines were removed programmatically, leaving single Voronoi segments that encompassed the structures. While this does not strictly adhere to a true Voronoi implementation, it is considered close enough for the purposes of this comparison.

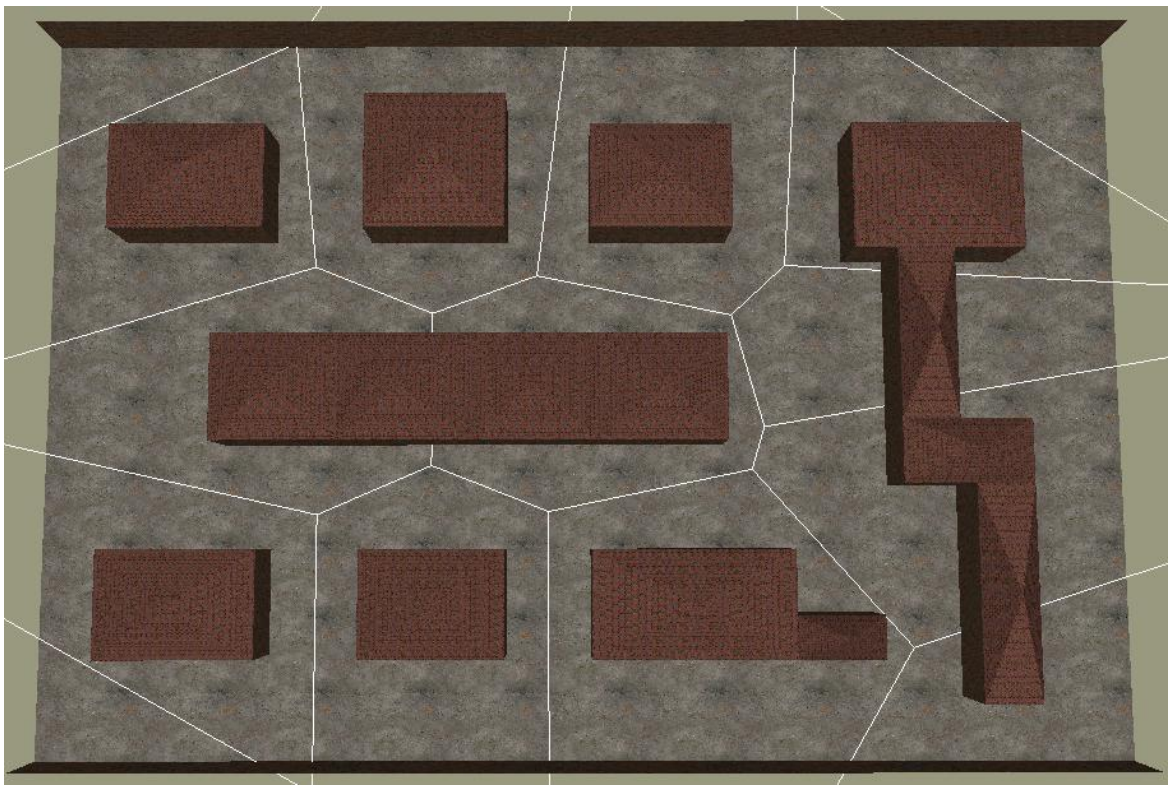

Figure 19. An initial implementation of the Voronoi graph, with multiple sites used for the large centre and right building.

The last remaining issue with the graph is one that is shared with the Visibility Matrix implementation. Due to the nature of the scene's design, the open edges surrounding the buildings do not contain any points or paths for agents to follow. The Voronoi segment boundaries that pass out of the scene converge into infinite lines, as no points are present for the algorithm to consider. As discussed in Chapter 5.1.1, a larger scale scene could potentially disregard the boundaries of the simulation, as the viewer's focus would be on the high density centre of the scene. In such a situation, agents could be removed when leaving the scene and be repopulated at another edge or position in the scene. However, in this simulation, as with the Visibility Matrix implementation, agents are required to remain inside the scene, so the decision was made to add a perimeter line around the Voronoi graph, to close up the existing segments and keep them inside of the scene's boundaries. The result of this can be seen in Figure 20. All trailing segments edges were ignored, and the final points and paths entered into a graph that could be read by Dijkstra's in the same format as the Visibility Matrix.

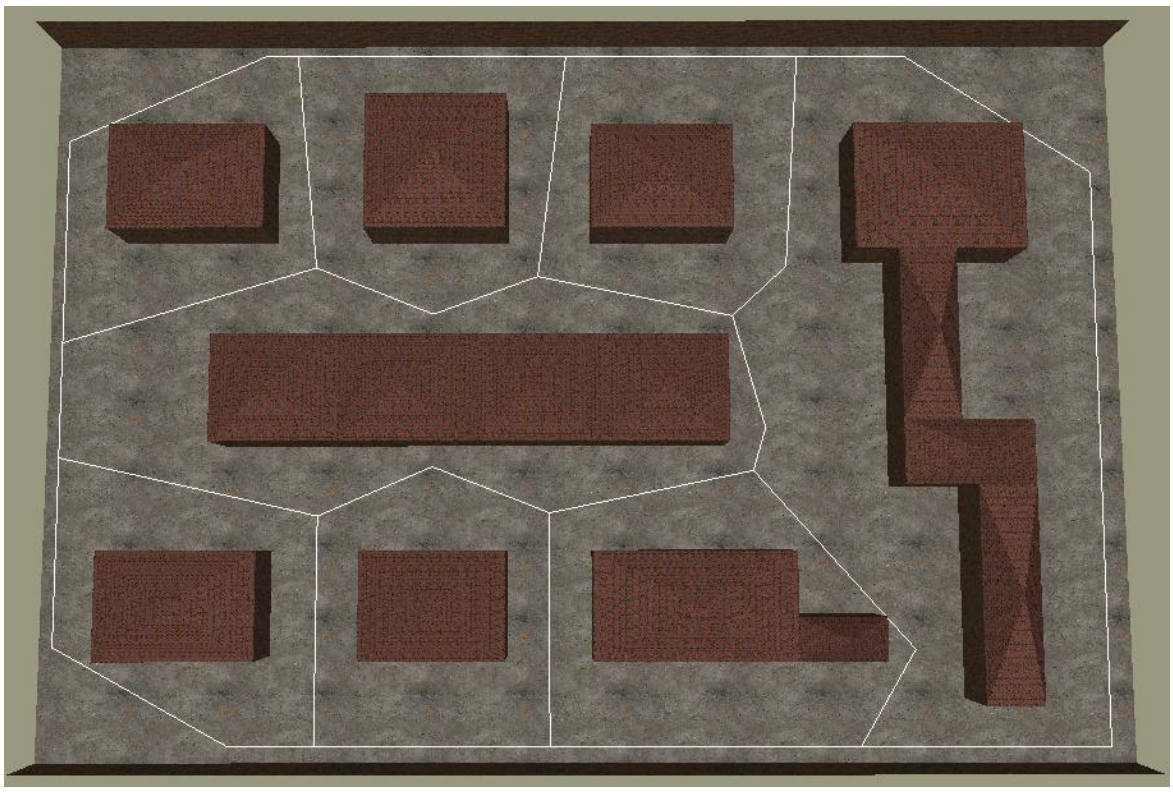

Figure 20. Capping Voronoi segments at the boundaries of the scene provides a close system for agents to navigate.

5.1.3 Dijkstra's Algorithm

With both the Visibility and Voronoi routing arrays complete, the framework was ready to have Dijkstra's shortest path algorithm applied to navigate paths for the agents. Given a starting and target node from the array in use, the algorithm determines which set of points provide the most efficient route between these locations, and returns this as a sequential sequence of numbers. Vectors are calculated by using the agent's current location and the coordinates of the next location in the path list. Once the agent has reached its location, the next path point is selected and the vectors are recalculated for this new path. This continues until the final node is reached, at which point a new target is selected, and the cycle is restarted with a new list of calculated path points. A detailed flowchart of this is shown in Chapter 3.2.1, Figure 10.

The algorithm code was sourced from [Dijkstra, 2006]. As with the Voronoi source code, it required some adjustment to fit into the existing framework, to use the pre-generated path matrices as input parameters. The class controlling the agents calls the algorithm, passing in a start and end node. This returns an array containing points that represent the steps in the path. This is assigned to the current agent requesting a new route. Caching the path in this fashion means that the algorithm is called only when a new path is required, and not as each node is reached, which removes unnecessary computational overhead at

runtime. The complexity of the algorithm is defined by the number of points or vertices *V*, and the number of edges *E* joining these points, and as such, its running time is *O(E+V)*. In very large scenes, navigating a worst case path between the two furthest nodes could begin to make a measureable difference upon runtime efficiency. This would be especially noticeable in a densely populated scene in which many agents require new paths per frame. As described in Chapter 2, a proposed method to counter this would be to pre-cache all possible paths at compile time. If Dijkstra's shortest path algorithm is used to calculate every plausible route given a pair of start and end nodes, these path results could be stored in a corresponding array. At runtime, when an agent requests a route between two nodes, the pre-calculated path could be returned by querying the array of routes. The runtime overheads of this would be the time required to query the route array, which would be substantially less than calculating the route from scratch. Compile time would be increased substantially however, and memory requirements to store the routes could become an issue if the routing array was larger than the system memory available to the program. Provided resources were sufficient, this could be a valid method of counteracting the increased complexity posed by larger scenes of routing paths at runtime. In this simulation, such measures are not required due to the number of path nodes and edges, and all paths will be generated as required by the agents.

5.2 Collision Avoidance

At this stage of the implementation, agents were moving between locations, along the paths specified in either the Visibility or Voronoi Matrix. Many agents occupy the same paths, and inevitably the same position as they pass. The next stage of path finding was to create low level steering to move the agents out of each other's paths. These interrupt the high level steering trajectories and manoeuvre the agents on a new path until the collision is avoided. From here the agent is placed back onto a path towards their next target.

5.2.1 Proximity Detection

The proximity detection method requires an agent to have an area of 'awareness' around them that is used to trigger a collision detection response. This area is represented by a circle around the agent, with the circle's radius approximately twice the diameter of an agent model. Detections are performed between each agent's circle of awareness by comparing whether they intersect each other. If an intersection is detected, the agent being tested deviates from its assigned path to avoid the agent that is about to collide with him. At the next frame, the collision is retested and depending on the result, the agent is

either kept on an avoidance path, or redirected back towards its original target. Figure 11, in Chapter 3.2.2 explains this dataflow fully.

Implementation began with creating a collision detection class that could be passed the locations of two agents and return a collision status. The difference in *x* and *y* between the two locations is separately calculated, with the result of each squared, and then added together to produce a Comparison Range. This value is compared to that of the minimum collision range, squared. By comparing the squares of both values, the need to square root each answer is removed, reducing the computational complexity of the class. If the result of the Comparison Range is less than that of the test range, a collision has occurred. Once a frame, each agent is tested against every other agent in turn. If a collision is detected between the agent being tested, agent *a*, and another, agent *b*, the testing halts and enters into an avoidance stage. Initially, there is no indication of the direction to the collision only that it has happened within the predefined range. To calculate this, the dot product of the two agents is equated, giving the angle in radians of agent *b* in relation to agent *a*. It can now be determined whether the approaching agent *b* is to the front, side or rear of agent *a*. In this simulation, all agents will be moving at the same rate, which allows assumptions to be made about which collision detections are likely to culminate in a collision, if left unchecked. Rear and side detections should resolve themselves without the need to adjust either agent's trajectory. These will be due to sections of the scene that contain crossing paths, or when agents are following each other closely on the same path. In a real world situation, a human would not respond to somebody approaching in these directions as they would not be within their FOV. A preliminary test supporting this involved two agents walking in the same direction, just within detection range. The leading agent reported the collision in each frame, and attempted to enter the resolution phase. Had he ignored the trailing agent, the pair would have carried on without incident, proving that resolution was not required. By ignoring all detections that occur outside of an 80 Degree arc in front of the agent, the response system is greatly streamlined and represents a more realistic approach to the solution.

Assuming that a collision is likely, the angle of approach needs to be calculated so that the avoiding agent can move appropriately for the situation. This is done using the cross product of the two agents' path vectors, with the result indicating whether the collision is to the left or right of the agent *a*. Collision avoidance is only performed by agent *a*, as he is the agent being tested. Depending on the direction of agent *b*, *a* is calculated a new path that is 20 degrees left or right of its current path, and is instructed to follow this when the function to move all agents is executed later in the frame process. When agent *b* is tested for collisions, it too will calculate a route away from *a*, meaning that both agents should

pass each other successfully. At the end of the frame execution, all agents' collision statuses are reset, and any that were forced to avoid a collision have their trajectories calculated back towards their original goal. In the next frame, if the collision is still not resolved based on their new vector, the collision avoidance routine is started again. By resolving collisions in this way, all agents are assured of only moving as much as necessary to avoid a collision, and do not end up following long avoidance paths despite having potentially moved away from danger. Again, this approach attempts to simulate real life as much as possible, assuming that people will generally only deviate enough from their current path as necessary to avoid another person.

When tested with this level of implementation, agents were observed to bump off of an invisible space around each other, and only move from their path when there was very little space between the models. Similar behaviour has been observed on large scale implementations by other institutions, and can appear unrealistic on smaller, more detailed scenes. Increasing the size of the collision detection boundary around the agent would be an inefficient solution to this issue, as many more erroneous side and rear collisions would be registered in a densely populated situation. Instead, projected future positions of each agent are calculated and stored at each frame, simulating the location of the agent in 40 frames time. This allowed potential collisions to be avoided long before they were due to occur, with agents adjusting their course less aggressively and passing each other side by side. This shearing effect of agent movement gave a considerably better look of realism, as agents passed each other without appearing to have adjusted their path at all, due to only requiring minor path adjustments. The proximity detection code was adjusted so that agents checked for collisions both on their 'future self' and on their current locations, so any close range collisions that occurred due to path crossings or corners were still dealt with as originally designed. Some agent 'bumping' was still occasionally occurring on sharp corners (90 Degree or more). As two agents met on a corner in different directions, they would enter immediate collision resolution for close range detections. This was deemed acceptable as in real life scenarios, people cannot see through corners either, and are likely to have a close passing in a similar fashion, or even bump into each other.

5.2.2 Grid Method

The grid method requires the scene to be divided up into equal sections that represent the scene in a lower resolution, digital style format. Each cell is the size of an agent and can be one of two states; either occupied or empty. As agents move around the scene on routes set by the Dijkstra's path finding algorithm, they assign the grid location they

currently reside in as occupied, and continue changing the status of each grid space as they enter or leave them. An agent need only check if the grid cell he plans to occupy is free, and choose another if not.

Much of the calculation for the Grid method was abstracted out to the pre-processing section of the simulation, allowing for lower overheads at runtime. The main portion of this consisted of creating the grid and pre-populating the cells that contained buildings, with an occupied status. The grid itself is a two dimensional array, equal to the width and height of the scene, divided by the width of an agent, which produces over 900 cells. By using coordinates of the grid's first point (top left of the scene), and using the width of each cell, any given scene location, supplied in the form of *x y*, can be translated into a corresponding grid point. Using this methodology, the structures in the scene were translated into the grid so that they could be considered in collision avoidance calculations later on. By querying each cell in turn and testing its location with that of the scene's building structure lines, the cells were assigned occupied or empty statuses, to produce a grid ready for use with the agents. This is shown in Figure 21. As the grid preparation is calculated in the pre-processor, it ensures that the method is scalable for larger simulations, having no direct effect on runtime FPS.

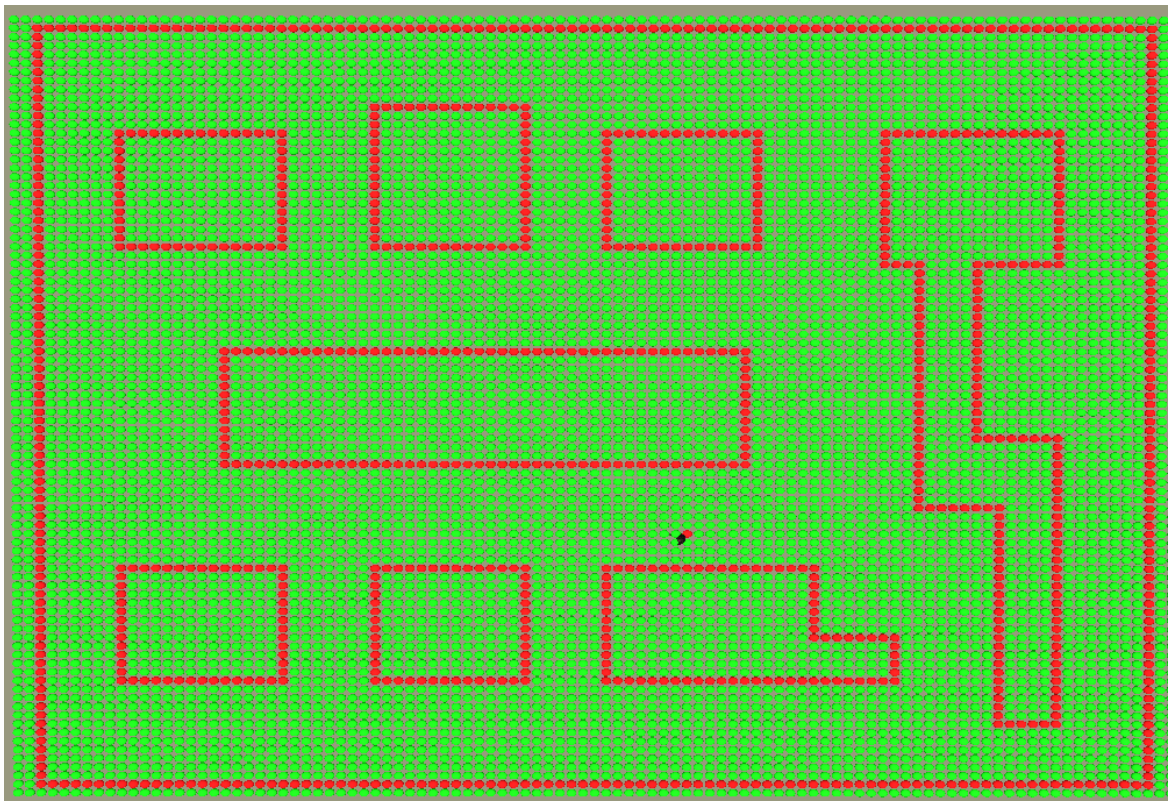

Figure 21. The simulation translated into grid cells and rendered with spheres to represent the status of each cell. A single agent travels through the scene, setting the status of its currently occupied cell as it moves.

Using a similar method to populating the scene with the building locations, a function was created that could return a grid location when passed a set of scene coordinates. At each frame, an agent calls this method to return its current grid location. With this obtained, the agent can update the grid with its location, marking it as occupied, and clearing any previous grid cells for that agent. As seen in Figure 21, the animation of the agent is accompanied by a red square, showing its location in the grid. In place of the proximity collision detection system, each agent checks the cell they are planning to move to within that frame. Using this method, no checks are required against other agents, as the grid keeps track of all agent and structural locations on a macro scale, meaning that collision detection is significantly faster than that of the proximity method. If the cell to be entered is occupied, the agent checks the left and right adjacent cells. If these are also occupied, the cells to the left and right of the agent are checked, followed by those behind him until a free cell is found. Reversing the coordinate to grid method described previously, a function was created that could return the coordinates for the centre of a chosen grid cell. Once an agent has found an empty cell during collision avoidance, this method is called and a route is calculated towards the centre of that cell. When the agent has moved to avoid the collision in the cell ahead of it, its path is calculated back towards its original route target that was supplied by Dijkstra's algorithm.

When executed, the reoccurring problem encountered when testing the proximity method was once again apparent, with agents aggressively changing route, or stopping suddenly in front of one another. This was calmed by detecting collisions both further ahead of the agent (four cells worth of distance), and directly in front of it, similar to the solution used previously. This allowed for any unforeseen collisions created by a change of direction or the crossing of paths to be dealt with, while providing a more foresighted approach for future dangers. Some sharp changes in direction were still observed, and were concluded to be due to the nature of the grid method, and because the scene was being handled in a lower resolution to that of the proximity method. As buildings were included into the grid's calculation at compile time, agents were able to make more informed avoidance decisions when posed with a potential collision. Another drawback observed was the algorithm's difficulty in dealing with dense build ups of agents, and was especially noticeable when applied with the Voronoi graph. This is again due to the low resolution of the scene, being represented by fixed state cells, allowing fewer agents to populate a set area of space. This will be further analysed within the Results Chapter of this thesis.

6 Results

This results chapter will measure each steering method for computational efficiency, and visual quality of realism, to ascertain the best combination of high and low level techniques. This is achieved in part through user testing, where a selection of individuals were asked to choose combinations of behaviours that they deemed the most realistic in appearance.

6.1 Efficiency of Methods

With each high and low level steering method programmed in isolation within its own class, the simulation allowed for simple, clean measurement of each feature, without the danger of cross contamination by other classes. As specified in the objectives of this thesis, each implementation would be compared by its use of computer resources during the pre-processor start up of the simulation, and during runtime, where inefficiency would take a toll upon frame rate. As stipulated previously, the primary goal of this simulation is the behaviour of agents, and consequently, the graphical element of the final implementation has not been focused on as much as the agent steering. To factor this into the testing, each method was tested both with, and without a graphical element being rendered. In tests without rendering, agents are moving and interacting on a purely mathematical plane. The strength of the graphics hardware in the test machine is no longer a consideration, and the test can be described as purely AI focused. It is accepted that certain features of the program framework could still affect the test results, such as object handling or memory control techniques, but as these will remain constant for all tests, they can be disregarded as not having an effect upon any singular method.

6.1.1 Frame Rate Analysis of Low Level Steering

The freeware program Fraps was used to measure frame rates of rendered scenes, but was unable to measure the calculation speed of non-rendered tests. These were measured by placing system timers within the code at the entry points of the section to be tested, and comparing the value recorded with a similar timestamp taken at the end of the function. Unrestricted by graphical bottlenecks, the times to compute each frame were almost immeasurably small by conventional methods of recording, on the test hardware. The smallest denomination of measurement for the timeGetTime() function is milliseconds (ms), so each test was repeated 1000 times allowing for a larger, more meaningful value to be obtained. This was then divided by 1000 and then converted back to milliseconds. As stipulated in the objectives, each test was repeated ten times so that an average could be obtained, and were conducted against five densities of agents; 100, 500, 1000, 5000,

and 10000. The results of each individual test are supplied in the Appendixes, and are summarised in Figure 22.

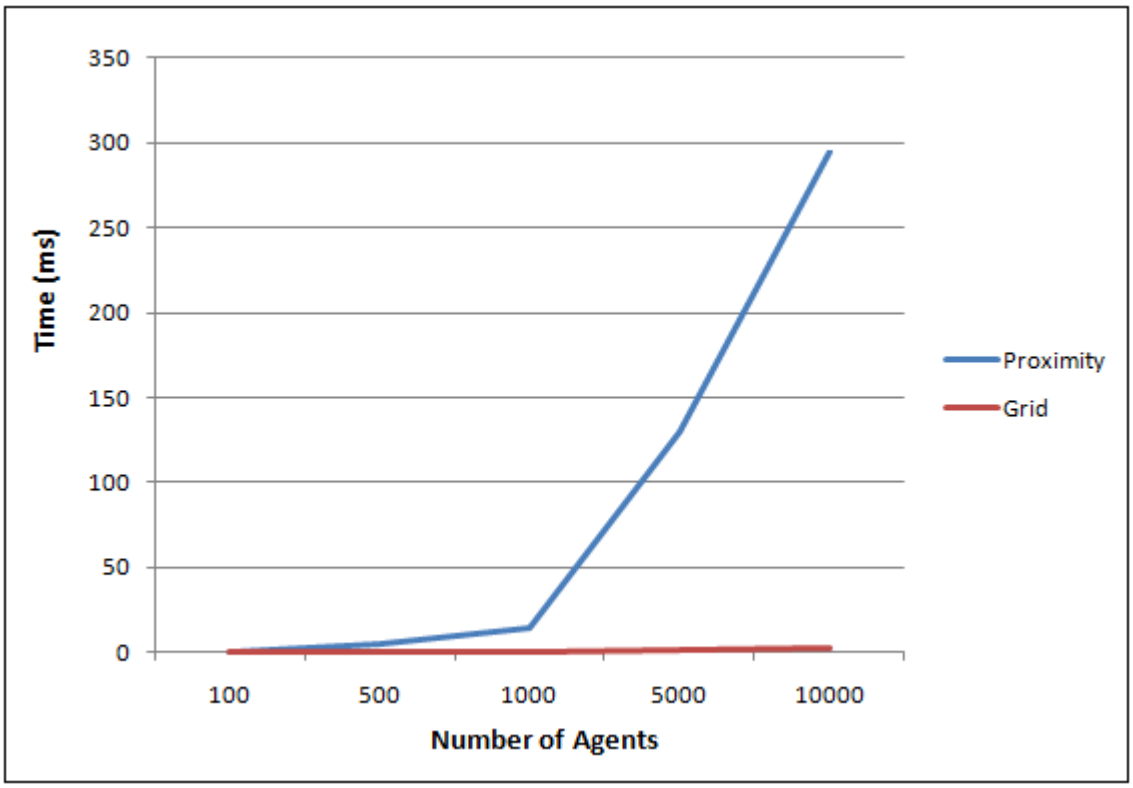

Figure 22. This graph shows the relationship between the number of agents in a scene and the time taken to process one complete frame (without rendering).

As shown in the graph, the proximity method drastically depreciates in efficiency when the agent total rises above 1000, reducing to processing a little over three frames a second with 10,000 agents. In the observations described in Chapter 4.4.2, the simulation must optimally aim to maintain a minimum of 25 frames per second, or risk detrimental effects to the AI due to the drop in sample rate for collision testing. Using the results displayed in Figure 22, this is reached with approximately ~1600 agents. The Grid method is almost unaffected by the rise in agent quantity, showing an increase of 3 ms between the lowest and highest densities, compared with the 295 ms increase of the Proximity method.

It should be noted that the theoretical maximum number of agents that the simulation can handle using the Grid technique is equal to the number of empty cells. This is determined by the size of the uniform grid used to divide the scene, minus the locations occupied by buildings, which approximates to ~4800 cells. However, a scene with this many agents would require every space to be populated, and the AI would come to a standstill as a collision resolution would not be achievable without free grid space to move agent into. For the purposes of these tests, collision resolution was deactivated, leaving only

detection testing, allowing for five and ten thousand agents to be measured using the Grid method.

Memory usage was found to be comparable between each method, resulting in the decision to disregard it as a measurable factor during testing. The only rises that were observed were when agent densities were increased, or when human models were used to represent agents. These were both due to the extra class objects that are created in processing these tasks and were not due to the methods of steering being used. Memory use during runtime was also constant across all implementations, with no rise in the initially reserved quantity. This signifies that no memory leaks were present, and all methods could be run indefinitely on the test hardware, from a resource perspective.

Fraps frame rate testing was conducted with both spherical, and human model representations of agents measured separately. The upper limits of the graphical hardware were first established by running the simulation with collision detection and resolution removed, leaving only path finding to move agents between global path nodes. Once again, the optimal frame rate of 25 fps was targeted, and each method was adjusted until this was reached. The values attained were 1500 agents using the spherical representation, and 37 using human models. Removing the scene's buildings had no measurable effect upon fps, which was expected as their combined polygon count equated to less than 5 spherical agents. Applying the Grid and Proximity methods to human models had no discernable effect, corroborated by Figure 22 which shows that they have very little impact on processing time when controlling such a small density of agents. As expected from the non rendered test results, the Grid method had no effect upon a scene using spherical agents either, but the proximity method lowered the fps by 26%. This equated to a drop of 400 agents to reach the 25fps target, leaving a final achievable total of 1100 agents when using the proximity method with spherical agents.

6.1.2 Pre-processor Analysis of High Level Graphs

Both the Visibility and Voronoi graphs are constructed entirely in the pre-processing section of the application. Although this means that neither has an effect on runtime speed of the simulation, they were still timed and measured so that they may be discussed within the context of scalability to different scenarios. As seen within the Appendix (Table 1 and 2), both graphs compute the entire scene in less than 10 ms, with the Voronoi method computing the fastest at less than 1ms. These results show that either method could potentially be integrated into a runtime scenario with a small impact upon performance. Many other studies described in Chapter 2 use these methods in this way, such as

Champagne et al. [Champagne, 2005] who use Voronoi graphs that are generated per frame for spacing groups of agents. These will be further discussed in the Conclusion of this thesis.

The average cost of Dijkstra's path finding algorithm was 0.028 ms when used against the Voronoi Graph, and 0.030 using the Visibility Matrix. This is a very low computational overhead; even in a worst case scenario where every agent could request a new, long distance path, simultaneously. As expected, requests using the Voronoi graph were nominally less as it contains far less edges and path nodes compared to the Visibility Matrix.

6.2 Comparisons of Local and Global Steering Methods through User Testing

The second stage of testing involved quantifying each steering method based on a visual comparison to a real life scenario. The aim was to determine which high level and low level steering techniques looked best when combined. To measure this fairly and accurately, a selection of testers were asked to view short video clips of the simulation, rating them in order of how realistic they thought each looked. Four videos were created, each displaying a different combination of the two high level, and two low level steering methods. These simulation videos were accompanied by a fifth consisting of real people moving through a local shopping centre. This film was taken from a position overlooking a walkway that was roughly 4 meters wide, and of medium population density. Traffic can be seen flowing into and out of the scene from the top and bottom of the camera shot, causing opposite streams of people to navigate around each other on the way to their destinations. The conditions of the location, and the quantity of people meant that the steering behaviour exhibited in the pedestrians was an ideal comparison for the simulated videos. After being shown the video of real people, testers were presented with each of the simulation videos in a random order that differed between each test. This was to remove any influence one video may have had upon another's appearance due to display order. Once all videos were complete, the testers were asked to give comments on the methods in addition to ratings.

6.2.1 Rating Scores

Twenty individuals were shown the video selection, with their scores recorded and summarised in Figure 23. A breakdown of each score by person is available in the appendix. As the results show, the preference towards the visibility graph was significantly higher than that of the Voronoi. Combined, the Voronoi method scored 63 against 137 for

the Visibility Matrix. The difference between low level steering was less profound but still clear, showing the proximity method as the favourite.

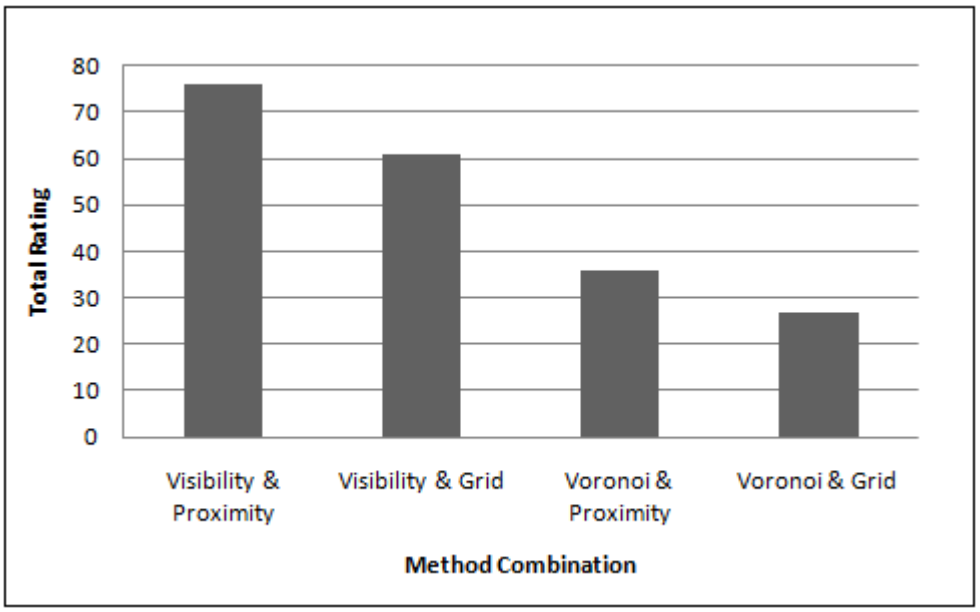

Figure 23. Results of user testing, ordered by ratings of realistic appearance

The order that each video was shown, appeared not to have affected the scores, with no discernable patterns arising. Users were asked not to concentrate upon any specific behaviour, but requested to give scores based upon the overall appearance and flow of the agents in the scene, while thinking about the real life equivalent. It was stipulated that the demonstration was targeting realistic looking macro movement of agents, not the interactions between them. It should be noted that users were not told how each method worked as this could have made the strengths or weaknesses of certain methods more obvious and influenced the result.

6.2.2 User Comments

As anticipated, testers were quick to note the lack of human interaction between agents, showing a natural expectation of more detailed behaviour, with many comments revolving around this subject. As discussed, human interaction is not a focus of this study, although it is a logical step forward from this research and will be further covered within the Conclusion Chapter. Other generalised comments mentioned that no agents were seen to stop or slow down, therefore becoming obstacles themselves within the scene. A lack of obvious targets was noted by some testers with one liking it to observations from free roaming city based video games, such as Grand Theft Auto IV. He described that in such titles, pedestrians appeared to walk with purpose but without embarking from, or arriving at any specific destination. If agents were seen to emerge from and disappear into certain

buildings from the scene, with more noticeable quantities of people visiting these locations than others, it would improve the overall scene aesthetic.

Testers were asked to describe why they had chosen one method over another, putting their answer in the context of a comparison to real life. With regards to the Visibility Matrix, preference was generally shown due to the wider spread of people, and the better use of the scene's space. Many testers noted that when they wanted to move from one location to another while navigating a course of buildings, they would attempt to take the shortest path around each obstacle, in a similar fashion to the way in which the Visibility Matrix method steered agents tightly around buildings and corners. It was with this reasoning that the Voronoi method was looked upon less favourably, due to the wide berth that agents were affording each building. Additionally, the Voronoi method was noted for having fewer paths, and showing tighter agent groupings, however one user noted that this gave the agents an appearance of moving with a common purpose as if travelling towards a destination of shared interest. Other notable comments regarded the bunching of agents observed at waypoints, which affected both methods.

When asked to differentiate between low level steering methods, answers were less forthcoming, as people had been more drawn to the movement of the agents as a whole. Repeated observations did highlight that agents using the grid method appeared to bump into each other more frequently, and seemed more mechanical in their movement due to last minute collision avoidance. Some testers did notice that in areas of high density, the occasional flanking agents appeared to move through walls while avoiding each other (graphical clipping). This was not noticed on the Grid demonstrations, as agents using this method are aware of buildings when dealing with collision resolution. It was also suggested that to better impersonate real life, and especially in more dense areas of the scene, agents should slow down when avoiding multiple collisions.

6.3 Summary of Findings

Computational analysis and qualitative user testing present very different results to the question of which methods are best suited for crowd steering. It is clear from user testing that the strongest combination is the Visibility Matrix and the Proximity method; however this displayed crippling computational cost when scaled for use on large crowd densities. The underlying reason for this inefficiency is because of a decision to have each agent check every other agent per frame, for collision proximity. Even with optimisation, the Proximity method would not be as efficient as the Grid collision detection, which was consistently fast despite vast increases of agents. However, as described, testers did not

look as favourably towards the grid method due to its blocky appearance when dealing with collision resolution. When dividing a scene into grid cells and making agents adhere to these, the level of motion smoothness found with analogue methods such as the proximity test will not be achievable in a comparable fashion.

Both high level path creation techniques were well suited to Dijkstra's algorithm, and shared similarly fast navigation times. Neither was noticeably preferable based on processing time alone, but as qualitative testing proved, the Visibility Matrix created a more realistic path map of the scene. This was mainly due to the spread of routes, and their quantity, creating a scene that was evenly populated, while still showing areas of increased traffic. The limited paths created with the Voronoi method, and their distance from each structure, gave a crowded and unrealistic look to the scene, suggesting that a less dense population may have better suited this method. Consequently, the increased flow of agents per route gave an increase in the number of collision avoidance calculations per frame, which in itself is an undesirable outcome.

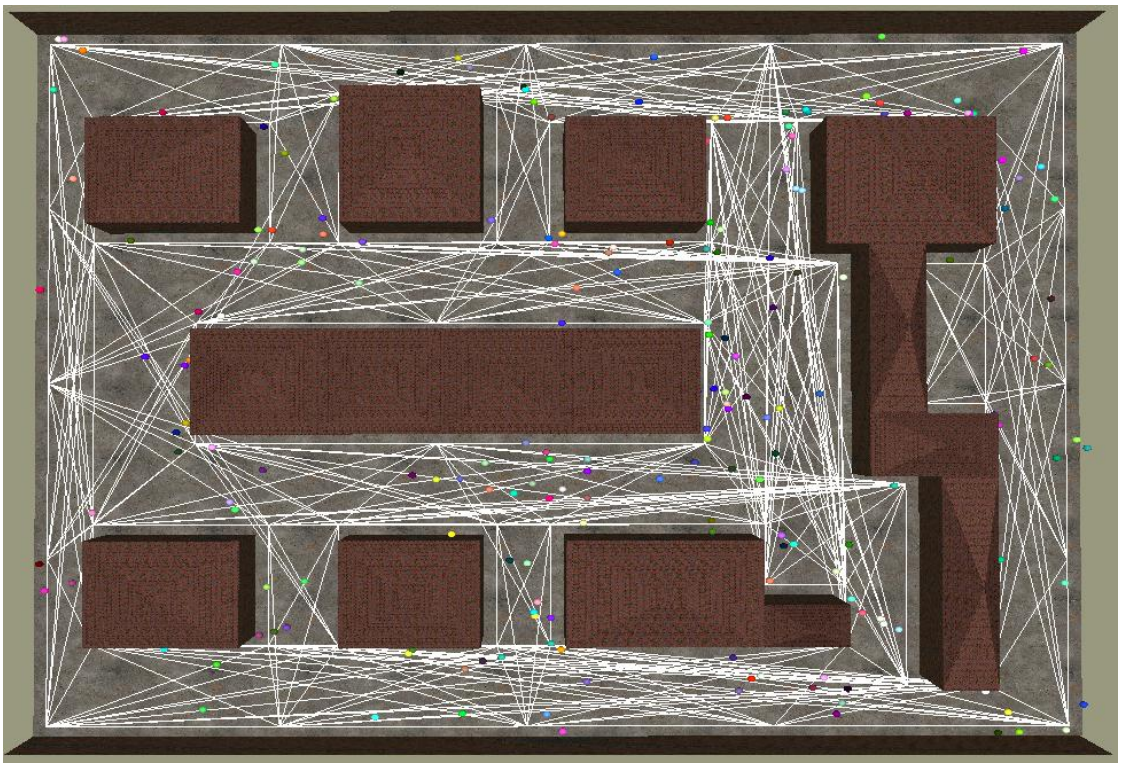

Figure 24. The final implementation showing agents (represented by spheres) following the paths created by the Visibility Matrix method.

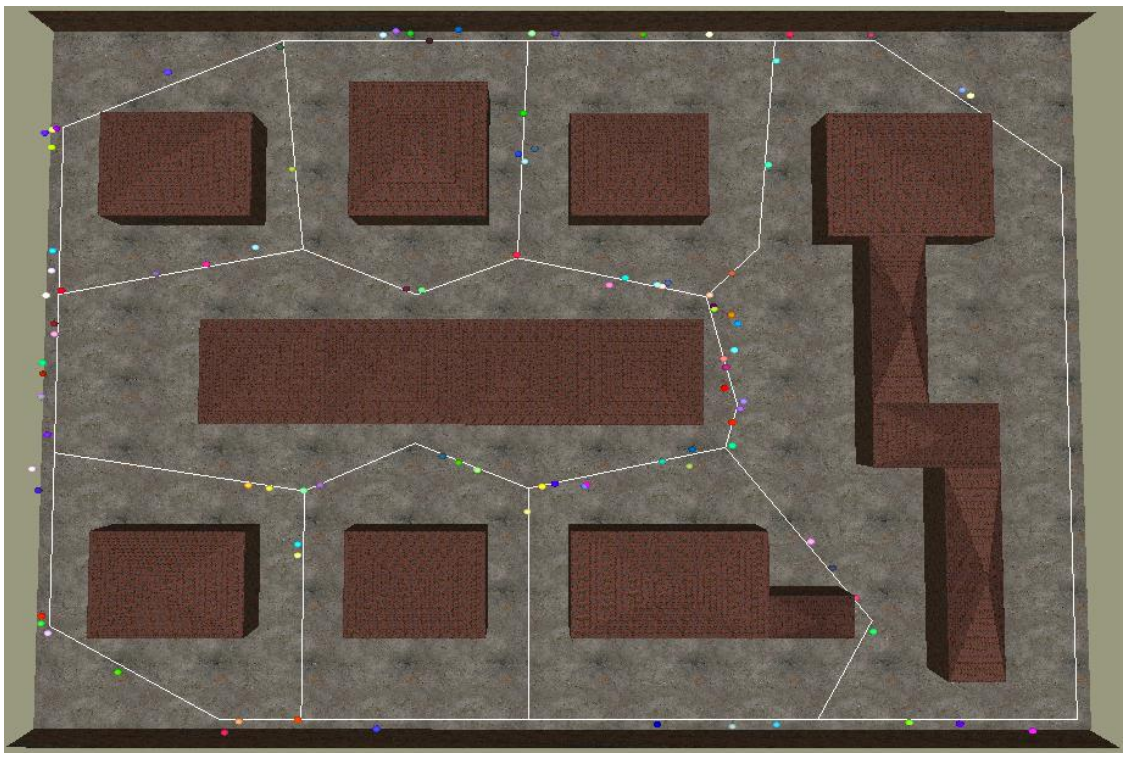

Figure 25. The final implementation showing agents (represented by spheres) following the paths created by the Voronoi Graph method.

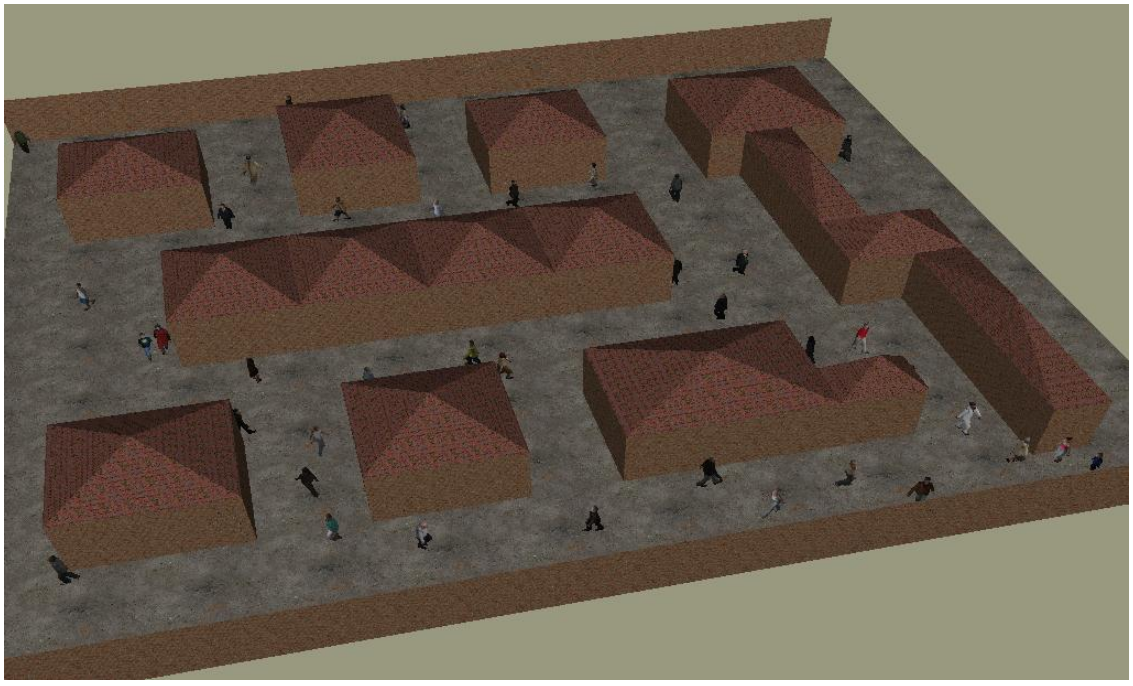

Figure 26. Agents navigating the scene using paths created by the Visibility Matrix method.

7 Conclusion

This chapter will discuss the success of the project, and areas for possible future work will be outlined.

The aim of this thesis was to present a comparison of two popular high level, and low level steering methods, in an attempt to suggest the ideal combination for a reusable solution. Each method offers its own pro's and con's over its counterpart which became clear during implementation and testing. Of the two high level steering techniques compared, the Visibility Matrix was more favourable over that of the Voronoi graph. The greater amount of paths generated by the Visibility Matrix allowed for a much more even spread of agents around the scene, as opposed to the densely populated and sparse spread of paths generated by the Voronoi method. This offered advantages in both the visual realism of the overall movement and flow of agents, and the lesser amount of collision detection testing required due to the greater spread of agents. The Proximity method of collision detection proved to be a more effective low level steering method than that of the Grid based method in this implementation. Failures in the Grid method were apparent if collisions occurred simultaneously with agents migrating from one cell to the next, with the problem worsening at path nodes where many agents were moving to a common goal. Although the detection and avoidance of the Proximity method was superior, it suffered great computational drawbacks when applied to larger crowds, an issue that is addressed in the following section of this Chapter regarding improvements. As presented in the Results Chapter, the combination of high and low methods that complimented each other best were that of the Visibility Matrix and the Proximity Method. The results gave clear indicators that although methods can be praiseworthy from a computational standpoint, they can still lack the detail and realism required when measured qualitatively.

7.1 Improvements and Further Work

Further research and expansion on the work presented in this thesis would allow scope to compare more methods of high and low level steering presented by other papers. Although this thesis was limited by a finite selection of methods, many of the papers referenced present their own solutions that would complement the research already undertaken here. Some researchers appear to be moving to use the A* path finding algorithm for high level steering as an alternative to Dijkstra's. Generally, the computational cost of the A* algorithm is less then Dijkstra's while achieving the same result in a similar time. This is achieved by using a heuristic estimate of the final path length to steer the algorithm in its choice of nodes to search. However, in complex cases Dijkstra's path finding algorithm is more effective at locating the target node. Applying A*

to the implementation presented here may offer better computational results, or worse depending on the circumstance, and therefore it would be a worthwhile further investigation. Similarly, the method of 2nd order Voronoi segmentation described by Sud et al. [Sud, 2007], could be used to better segment the scene for high level path finding. Alternatively it could be used in conjunction with the research presented by Champagne et al. [Champagne, 2005], to steer the crowds on a low level as collision detection and avoidance. This was also described previously in Chapter 2 during the literature review.

Following on from the computational results presented in Chapter 6, it is clear that optimisation is needed for the Proximity Method to be considered as a viable solution for large simulations. Had a method of area segmentation been implemented to overlay the scene and divide agents into groups of local neighbours, collision tests would have only been needed between individuals in these local groups. The cost of each frame computation would be dramatically decreased using this method, allowing for a higher number of total agents. To add this feature to the existing implementation, area segmentation could be achieved either by division of the scene into zones containing many paths, or by using the paths themselves as the method of subdivision.

Dividing the scene into arbitrary sections is best done procedurally to maintain a scalable crowd solution that does not require manual input. This can be done using the Voronoi solution already implemented in this thesis, but with the focus of interest changed from the edges and nodes to the Voronoi segments. Figure 24 reuses the example of a Visibility Matrix and a Voronoi Graph displayed in Figure 7, but overlays the two to display how the Voronoi can be used to subdivide the scene into segments. Using this method, tests for collisions between agents can be restricted to those that share the same Voronoi segment.

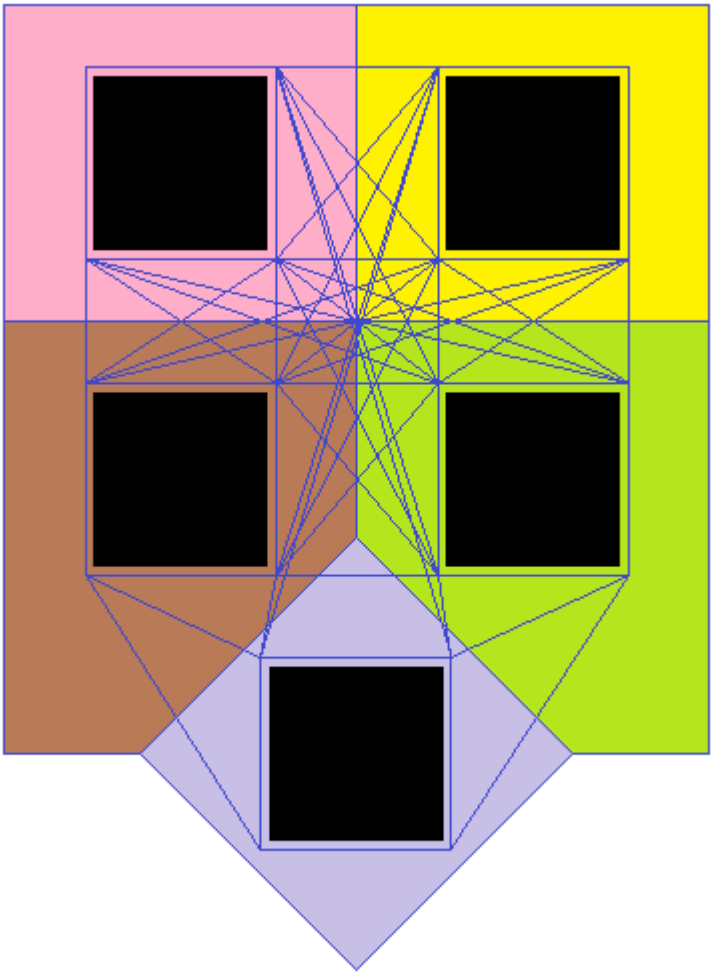

Figure 27. Using a Voronoi Graph to subdivide a given scene that already contains waypoints for agent navigation.

The drawback of this is during the transition of agents between segments there is the possibility of agents being within a close proximity of one another, but classed as within different Voronoi segments. This is clearly noticeable in Figure 24 where the centre of the pink, yellow, brown, and green segments join. This also happens to be a busy waypoint crossover, meaning that the chance of it being densely populated by agents is quite high. To overcome this shortfall, collision detection areas could be expanded to include adjacent segments to that which the agent currently populates, allowing for the passing of segment borders to no longer be a concern. As this would increase the quantity of agents to test, thus lowering the efficiency once more, it would require that the segments were decreased in size. As discussed in Chapter 2, Sud et al. [Sud, 2007] uses second order Voronoi diagrams to further segment the scene, creating a Voronoi graph where each segment is closer to one of a pair of sites than to any other site. Figure 25 shows the example given in Sud's paper.

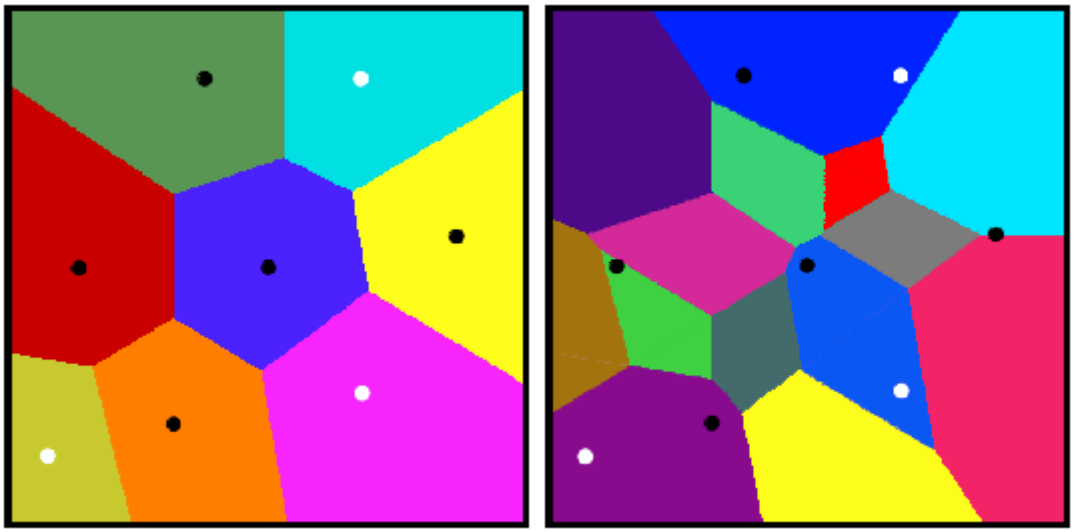

Figure 28. Further processing a Voronoi Graph so that the second order is calculated, returns a greater amount of segments that can be used for scene subdivision in collision avoidance. Adapted from [Sud, 2007].

The alternative method of scene division is to use a similar principle on the waypoint paths, testing only against agents that share the same path, or adjacent paths. This ensures that agents approaching a waypoint will be aware of other agents arriving on a different path. This is displayed in Figure 26, showing the path that the agent currently occupies, and the adjacent paths that would be tested for collisions. Further efficiency improvement could be achieved by dividing the paths into sections, negating the need to test against agents at opposite ends of long paths. This could also be extended to reduce redundant testing against agents on adjacent paths that are great distances from each other in terms of a potential collision risk.

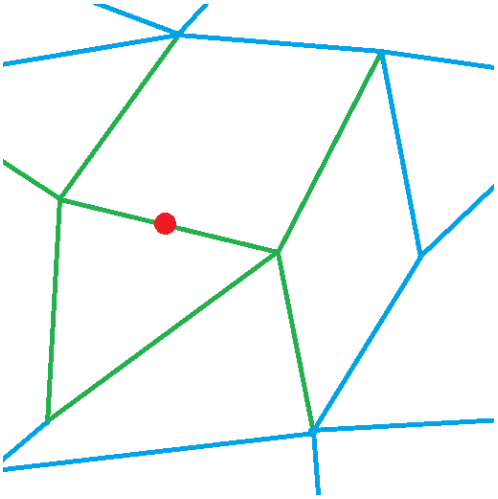

Figure 29. Proximity testing can be made more efficient by restricting collision tests to the path that an agent is on, and adjacent paths jointed by path nodes. The agent is represented as a red dot, with the paths to be considered in testing coloured green. Paths to be ignored are blue.

An improvement that would benefit all high level steering methods is the way in which waypoints are dealt with during agent navigation. Although more noticeable when using a Voronoi Matrix due to the low number of paths, agents are seen encountering difficulty when approaching densely populated waypoints. This is due to a conflict of instructions; collision avoidance is steering the agent away from or around the local mass of agents, but as soon as the collision is cleared, they are returning back towards these agents, in an attempt to reach the waypoint and mark their path as complete. If larger waypoint areas could be used that the agent need only get within range of, it may reduce the pressure on the agent to reach a smaller target.

One of the main observations by users during testing was the lack of agent animations depicting interactions between them. Key suggestions involved variations to the way in which agents pass one another. Users noted that in tight spaces, real life people will tip their body towards or away from a passing person, reducing their width temporarily to fit through the space. It appears from testing, that a logical progression of the solutions presented in this thesis would be to implement a number of complementary animations such as this, if the increase to realism was worth the extra computation overhead.

One last improvement to the decision making of the agents concerns the way in which agents choose targets. As discussed in Chapter 6, one user noted that agents did not appear to travel from or too any singular location specifically, instead, agents moved continually around the scene in a mindless fashion. A solution to this could be to relocate certain waypoints to centre on building walls, representing doorways. As agents reach these waypoints, they could be withdrawn from the simulation temporarily, signifying that they have entered into the structure, only to return later after an arbitrary time. By weighing these waypoints more favourably when Dijkstra's shortest path algorithm is choosing a target location, the flow of agents targeted to these building entrances could be set at a desired level, showing a common interest across the crowd as a whole. By removing or changing between favoured locations during runtime, in a similar fashion to that seen in [Pelechano, 2007], the flow of agents would maintain a level of variation across the scene.

To improve the scene structure layout, it has been considered that the building wall and roof creation algorithms presented in Chapter 4 could be further extended to generate randomly sized structures procedurally. This would allow new and diverse scenes to be created in a fraction of the time manual design takes, and would offer an impressive showcase to further test the scalability of each steering method.

8 References

[Arikan, 2001] Arikan, O., Chenney, S., Forsyth, D.A., (2001). Efficient Multi-Agent Path Planning. *Eurographics Workshop on Animation and Simulation*.

[Byszewski, 2009] Byszewski, P., (2009). Real-Time Visualization of the Human Evacuation Algorithm. *13th Central European Seminar on Computer Graphics.*

[Champagne, 2005] Champagne, J., and Tang, W., (2005). Real-time Simulation of Crowds Using Voronoi Diagrams. *EG UK Theory and Practice of Computer Graphics*.

[Choset, 1995]Choset, H., and Burdick, J., (1995). Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph. *Robotics and Automaton*, vol. 2, pp. 1649 – 1655.

[Dijkstra, 2000] Dijkstra, J., Timmermans, H.J.P., and Jessurun, A.J., (2000). A Multi-Agent Cellular Automata System for Visualising Simulated Pedestrian Activity. *Theoretical and Practical Issues on Cellular Automata, pp.* 29 – 36.

[Dijkstra, 2006] Dijkstra's Shortest Path Algorithm in C++. http://ds4beginners.wordpress.com/2006/12/17/code-2-for-dijkstras-shortest-path-algorithm-in-c/

[Dijkstra, 2011] Dijkstra's algorithm: http://en.wikipedia.org, *referencing:*

Dijkstra, E. W., (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, vol. 1, pp. 269–271.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., (2001). Dijkstra's algorithm, *Introduction to Algorithms*, Second edition, Section 24.3, pp. 595–601.

Zhan, B.F., Noon, C.E., (1998). Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science*, vol. 32, no. 1, pp. 65–73.

[Fortune, 2011] Fortune's Algorithm in C++, by Matt Brubeck, http://www.cs.hmc.edu/~mbrubeck/voronoi.html

[Foudil, 2006] Foudil, C., and Noureddine, D., (2006). Collision Avoidance in Crowd Simulation with Priority Rules. *European Journal of Scientific Research*, vol. 15, no. 1, pp. 6-17.

[Haciomeroglu, 2009] Haciomeroglu, M., (2009). Populating virtual urban environments with crowds of pedestrians in real-time. *Thesis submitted for PhD, at the University of East Anglia.*

[Hershberger, 1989] Hershberger, J., (1989). An Optimal Visibility Graph Algorithm for Triangulated Simple Polygons. *Algorithmica*, vol. 4, no.1-4, pp. 141-155.

[Hoff, 2000] Hoff, K., Culver, T., Keyser, J., Lin, M., and Manocha, D., (2000). Interactive Motion Planning Using Hardware-Accelerated Computation of Generalized Voronoi Diagrams. *Robotics and Automaton,* vol. 3, pp. 2931-2937.

[Low, 2007] Low, M.Y.H., Cai, W., Zhou, S., (2007). A Federated Agent-Based Crowd Simulation Architecture. *European Conference on Modeling and Simulation*, pp. 188-194.

[Mitchell, 2000] Mitchell, J.S.B, (2000). Geometric Shortest Paths and Network Optimization. *Handbook of Computational Geometry*, pp. 633–701.

[OpenGL, 2011] OpenGL Tutorial: Quaternion Camera Class by Vic Hollis, http://www.nehe.gamedev.net

[Pearce, 2004] Pearce, D., Ryder, G., Lapeer, R.J., and Day, A.M., (2004). Exploiting Partial Visibility for Optimised Crowd Scene Rendering. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pp. 34-35.

[Pelechano, 2007] Pelechano, N., and Malkawi, A., (2007). Comparison of Crowd Simulations for Building Evacuation and an Alternative Approach. *Proceedings: Building Simulation*, pp. 1514-1521.

[Reynolds, 1987] Reynolds, C., (1987). Flocks, Herds and Schools: A Distributed Behavioural Model. *Computer Graphics*, vol. 21, no. 4, pp. 25-34.

[Reynolds, 2006] Reynolds, C., (2006). Big Fast Crowds on PS3. *Proceedings of Sandbox (an ACM Video Games Symposium)*.

[Ryder, 2005] Ryder, G., and Day, A.M., (2005). Survey of Techniques for Rendering Real-Time Virtual Humans. *Computer Graphics forum*, vol. 24, no. 2, pp. 203–215.

[Silva, 2008] Silva, A.R., Lages, W.S., Chaimowicz, L., (2008). Improving Boids Algorithm in GPU using Estimated Self Occlusion. *SBC – Proceedings of SBGames'08: Computing Track.*

[Sud, 2007] Sud, A., Andersen, E., Curtis, S., Lin, M., and Manocha, D., (2007). Real-time Path Planning for Virtual Agents in Dynamic Environments. V*irtual Reality Conference*, pp. 91-98.

[Yilmaz, 2009] Yilmaz, E., Isler, V., and Çetin, Y.Y., (2009). The Virtual Marathon: Parallel Computing Supports Crowd Simulations. *Computer Graphics*, vol. 29, no. 4, pp. 26-33.

9 Appendix

	Computation to process Visibility graph (x1000)
Test 1	6345
Test 2	6361
Test 3	6435
Test 4	6393
Test 5	6408
Test 6	6354
Test 7	6341
Test 8	6417
Test 9	6367
Test 10	6394
Average (ms)	**6381.5**
Per Execution (ms)	**6.3815**
Per Execution (s)	**0.0063815**

Table 1. Results of the visibility graph creation test.

	Computation to process Voronoi graph (x1000)
Test 1	166
Test 2	154
Test 3	153
Test 4	152
Test 5	159
Test 6	154
Test 7	160
Test 8	156
Test 9	155
Test 10	165
Average (ms)	**157.4**
Per Execution (ms)	**0.1574**
Per Execution (s)	**0.0001574**

Table 2. Results of the Voronoi graph creation test.

	Computation to process Proximity collision (x1000) 100 people
Test 1	268
Test 2	266
Test 3	264
Test 4	266
Test 5	265
Test 6	262
Test 7	261
Test 8	267
Test 9	266
Test 10	267
Average (ms)	**265.2**
Per Execution (ms)	**0.2652**
Per Person (ms)	**0.002652**
Per Execution (s)	**0.0002652**

Table 3. Results of the Proximity collision test for 100 agents.

	Computation to process Proximity collision (x1000) 500 people
Test 1	4883
Test 2	4820
Test 3	4767
Test 4	4810
Test 5	4762
Test 6	4759
Test 7	4886
Test 8	4677
Test 9	4767
Test 10	4671
Average (ms)	**4780.2**
Per Execution (ms)	**4.7802**
Per Person (ms)	**0.0095604**
Per Execution (s)	**0.0047802**

Table 4. Results of the Proximity collision test for 500 agents.

	Computation to process Proximity collision (x1000) 1000 people
Test 1	14589
Test 2	14469
Test 3	14334
Test 4	13997
Test 5	14308
Test 6	14699
Test 7	14550
Test 8	14780
Test 9	14233
Test 10	14471
Average (ms)	**14443**
Per Execution (ms)	**14.443**
Per Person (ms)	**0.014443**
Per Execution (s)	**0.014443**

Table 5. Results of the Proximity collision test for 1000 agents.

	Computation to process Proximity collision (x1000) 5000 people
Test 1	128058
Test 2	126525
Test 3	130793
Test 4	129350
Test 5	129265
Test 6	129151
Test 7	133508
Test 8	131548
Test 9	128882
Test 10	128677
Average (ms)	**129575.7**
Per Execution (ms)	**129.5757**
Per Person (ms)	**0.02591514**
Per Execution (s)	**0.1295757**

Table 6. Results of the Proximity collision test for 5000 agents.

	Computation to process Proximity collision (x1000) 10000 people
Test 1	290744
Test 2	292371
Test 3	297471
Test 4	295821
Test 5	295232
Test 6	295715
Test 7	305544
Test 8	293489
Test 9	290993
Test 10	295245
Average (ms)	**295262.5**
Per Execution (ms)	**295.2625**
Per Person (ms)	**0.02952625**
Per Execution (s)	**0.2952625**

Table 7. Results of the Proximity collision test for 10000 agents.

	Computation to process Grid collision (x1000) 100 people
Test 1	17
Test 2	16
Test 3	17
Test 4	17
Test 5	16
Test 6	17
Test 7	17
Test 8	17
Test 9	17
Test 10	16
Average (ms)	**16.7**
Per Execution (ms)	**0.0167**
Per Person (ms)	**0.000167**
Per Execution (s)	**0.0000167**

Table 8. Results of the Grid collision test for 100 agents.

	Computation to process Grid collision (x1000) 500 people
Test 1	91
Test 2	93
Test 3	92
Test 4	92
Test 5	90
Test 6	94
Test 7	93
Test 8	92
Test 9	93
Test 10	93
Average (ms)	**92.3**
Per Execution (ms)	**0.0923**
Per Person (ms)	**0.0001846**
Per Execution (s)	**0.0000923**

Table 9. Results of the Grid collision test for 500 agents.

	Computation to process Grid collision (x1000) 1000 people
Test 1	235
Test 2	238
Test 3	231
Test 4	229
Test 5	235
Test 6	235
Test 7	230
Test 8	232
Test 9	231
Test 10	231
Average (ms)	**232.7**
Per Execution (ms)	**0.2327**
Per Person (ms)	**0.0002327**
Per Execution (s)	**0.0002327**

Table 10. Results of the Grid collision test for 1000 agents.

	Computation to process Grid collision (x1000) 5000 people
Test 1	1539
Test 2	1582
Test 3	1591
Test 4	1585
Test 5	1589
Test 6	1601
Test 7	1609
Test 8	1397
Test 9	1420
Test 10	1496
Average (ms)	**1540.9**
Per Execution (ms)	**1.5409**
Per Person (ms)	**0.00030818**
Per Execution (s)	**0.0015409**

Table 11. Results of the Grid collision test for 5000 agents.

	Computation to process Grid collision (x1000) 10000 people
Test 1	3152
Test 2	2944
Test 3	3197
Test 4	3213
Test 5	3221
Test 6	2967
Test 7	2790
Test 8	3060
Test 9	3144
Test 10	3197
Average (ms)	**3088.5**
Per Execution (ms)	**3.0885**
Per Person (ms)	**0.00030885**
Per Execution (s)	**0.0030885**

Table 12. Results of the Grid collision test for 10000 agents.

Tester		Visibility Matrix		Voronoi Matrix	
First	Surname	Grid	Proximity	Grid	Proximity
Glyn	Cotton	3	4	2	1
Isaac	Wilder	3	4	1	2
Anthony	Collison	4	3	1	2
Stephen	Howlett	2	4	1	3
Simon	Tovey	3	4	2	1
James	Maclean	3	4	1	2
Alex	Pooley	4	3	1	2
Craig	Scott	3	4	2	1
Mike	Neale	3	4	1	2
Ben	Ravenhill	3	4	1	2
Lorna	Ravenhill	3	4	2	1
James	Moore	4	3	2	1
Michael	Green	3	4	1	2
Ashley	Peters	2	4	1	3
James	Fenwick	2	4	1	3
Daniel	Wybrow	3	4	2	1
Nick	Richards	4	3	1	2
Michael	Price	3	4	1	2
Charlene	Mitchell	3	4	2	1
Simon	Whitley	3	4	1	2
Totals		61	76	27	36
Rating		3	4	1	2

Table 13. Results of User Qualitative Testing by tester.