Dynamic Object Avoidance.

by

Barry O Sullivan, B.A.Mod Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science

University of Dublin, Trinity College

September 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Barry O Sullivan

September 5, 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Barry O Sullivan

September 5, 2009

Acknowledgments

I'd like to say thanks to my friends, my family and my girlfriend for all their support. I couldn't have done this without you.

Barry O Sullivan

University of Dublin, Trinity College
September 2009

Dynamic Object Avoidance.

Barry O Sullivan, M.Sc.
University of Dublin, Trinity College, 2009

Supervisor: John Dingliana

This project will implement a generic system whereby large numbers of dynamic agents will be able to detect and react to dynamic world geometry. This system will be built on the foundation of a physics engine and will feature simple agents with basic behaviors. These agents will be given a sense of perception about their environment and the ability to avoid collisions with other agents and objects. This feature will be implemented via CUDA, a parallel computing architecture that enables developers to make use of the parallel nature of modern GPU's. These features will be tested on a variety of maps and situations in order to ascertain its viability as an addition to modern games.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years, games have benefited from the many advances in physics engines and middleware technologies, it is now difficult to find a game that does not employ some form of physics engine. The complexity of these engines has increased dramatically, with the majority of physics engines now capable of handling objects numbers in the thousands. Despite these vast improvements, little to no progress has been made in the field of agent awareness. Agents are as unaware of their environment as they were five years ago, they have no concept of dynamic objects. When these objects were sparse, this was rarely an issue, but it is now common place to find agents getting stuck behind dynamic geometry, especially as the complexity of these dynamic environments increases. These scenarios need to be avoided, they make the resulting game seem unprofessional and they break player immersion. Immersion is an integral part of any gaming experience, it lets players feel as if they are part of the game world. This increases the enjoyment and playability of said game. When this immersion is broken it can be quite jarring, resulting in players becoming uneasy. To address this, agents need to become more aware of their environment and react accordingly.

1.2 Dissertation Overview

This project will implement a generic system whereby large numbers of dynamic agents will be able to detect and react to dynamic world geometry. This system will be built on the foundation of a physics engine and will feature simple agents with basic behaviors. These agents will given a sense of perception about their environment and the ability to avoid collisions with other agents and objects. This is generally performed via a nearest neighbor query, but they are expensive to implement on a CPU, this is most likely why agents are not given proper understanding of their world as it is too expensive. This is why this project will attempt to port the nearest neighbor and avoidance code to the GPU via CUDA, a parallel computing architecture that enables developers to make use of the

parallel nature of modern GPU's.

This project will create a world with the following features:

- Individualistic agent modeling to simulate realistic, independent agents
- A physics engine to simulate a large, rigid body based world that includes both static and dynamic objects
- Vast numbers of dynamic agents and objects
- A robust description of the world for fast path planning and agent navigation
- CUDA implemented spatial divisioning algorithm for fast/efficient nearest neighbor queries
- CUDA based agent collision avoidance and anticipation

1.3 Project Layout

The layout of this thesis is as follows: Chapter 2 reviews the current state of the art within the field of agent simulation, specifically the techniques used to simulate large numbers of agents that react to their environment. Chapter 3 outlines the original design of the system. Chapter 4 details the implementation of this project, as well as the problems faced. Chapter 5 critically analyzes the implementation and discusses both the positive and negative outcomes of this system as well the results it offers. Chapter 6 presents the project conclusions and highlights possible future work that could be relevant to the project.

Chapter 2

State of the Art

Realistic crowd simulation is a heavily researched area. Over the years many diverging methodologies have been put forward, these range from looking at the problem on a per agent basis to looking at the problem as that of global crowd dynamics, where the actions of one agent are irrelevant. This chapter will give a brief overview of all the techniques available for multi-agent navigation, including techniques based on path planning and physics based agent dynamics.

2.1 Macroscopic Crowd Simulation

In macroscopic systems, a crowd is modeled as a whole, not a concentration of individual agents. Macroscopic crowd behaviors have been simulated in various ways. The earliest method is based on the equation for fluid simulation, where an agent is simply a component of a much larger Macroscopic system, such as a fluid of a gas. Others include generating density fields on a per area basis, which are then used to guide agents towards their goal.

Helbing [1] was one of the first to model the behavior of pedestrians as a gas-kinetic model. The underlying model is that of a fluid simulation, but the equations have been modified to handle agent characteristics. In fluid simulation a mechanism exists for reaching equilibrium, which occurs through the interaction process. For crowd simulation, this mechanism was altered so that equilibrium is reached when agents are moving at their desired velocity. This modification enabled Helbing to simulate dynamic crowd flows, which to this point had been difficult to achieve. Interestingly, emergent behaviors were also noted in this model, such as lane formation, which can be observed in congested pedestrian areas.

Bauer at al [2] uses a Macroscopic model to simulate crowds in public transport areas after large events. This model uses Helbings Macroscopic crowds for simulating crowd flows and for ascertaining likely areas of congestion. Helbings model has been modified so that agents have concepts of impatience and discomfort, depending on congestion and progression rate. This simulation was run on a

model of a train station and it simulated a massive influx of agents to the station. According to this paper, the simulation was effective, but lacked real world data to compare against their results. This work is of interest as public areas are often represented in games, especially during times of disorder and congestion.

Hughes [3] developed an alternative Macroscopic approach to Helbing. His paper presented a theoretical framework for understanding the movement of large crowds. This paper represents pedestrians (agents) as a continuous density field, which is driven by ay an evolving potential function that guides the density field optimally towards its goal. In this case agents are not modeled as discrete individuals, they are entities that optimize their behavior to reach goals within the density field. The theory presented does not govern the behavior of individual pedestrians, but the behaviors of large groups.

Treuille [4] presents a real-time simulation of crowds based on continuum dynamics, based on the work by Hughes. Motion is viewed as a per particle energy minimization with a continuum perspective on the system. Agents in this model have a discomfort field, which is a field they project around themselves, it represents the distance they would rather be from other agents if they can. The world is represented as a uniform grid, with a density field calculated for each cell. These cells are merged to give an overall density field. This model works under the concept of groups, at each update step it is only possible to calculate the movements for one group, the stipulation being that all agents in a group are going in the same direction. Obstacles are also included in the density field, so agents automatically avoid them. This method is capable of simulating 10,000 agents in real-time, but groups of agents can only travel in one of 4 directions.

2.1.1 Advantages of Macroscopic Systems

The main advantage of Macroscopic agent simulation is that it can simulate vast quantities of agents through relatively simple means, the governing equations can be easily modified to express different crowd dynamics. Through these means various types of games can be simulated, from RTS games (Real-Time Strategy) to heavily populated urban environments (Sand Box style games, eg. GTAIV). These agents are also able to avoid dynamic obstacles, which is essential.

2.1.2 Disadvantages of Macroscopic systems

The main disadvantage of Macroscopic agent simulation is the loose level of agent control, agents are not represented as individuals with their own goals but more as a member of a cohesive group. This loss of definition and control is highly undesirable in a gaming environment, as the majority agents must be completely autonomous and able to complete their distinct goals, without this adversely affecting the system.

2.2 Microscopic Crowd Simulation

In Microscopic systems, agents are modeled on an individual level. All behaviors and decisions are based on information spatially local to the agent, in other words, an agent scans its local environment and makes its behavioral decisions on the current state of the world. This model most accurately represents pedestrians and other autonomous agents. In a Microscopic system, the world is represented on an abstracted level that agents can easily interpret. Complex crowds behaviors, such as lane formation, are emergent and are loosely defined in these systems.

Lamarche and Donikian [5] presented an implementation of Microscopic crowds. This model is compromised of 4 parts, each working in conjunction. They are world representation, spatial subdivision, nearest neighbor computation and collision avoidance. This model is generic and can be used to simulate both indoor and outdoor environments. The geometric world is converted, using spatial subdivision, into a topological structure that agents are able to understand and navigate. Agents query their environment for objects and agents in close proximity. This information is used in a reactive navigation module for collision avoidance and smooth path following. Agents utilize the concept of personal space between agents in order to give realistic behaviors.

Shao and Terzopoulos [6] presented an implementation of Microscopic crowds within an urban environment. Like previous models, agents are entirely autonomous and base their decisions upon spatially local information. Agents are given motor, perceptual, behavioral and cognitive components. These components are integrated, resulting in individual agents that express realistic behaviors in urban environments. The world is represented as a hierarchal environment model and is compromised of perception maps, for nearest neighbor and obstacles queries, and paths maps, for path planning and following. It assumes that static objects remain static and that the majority of moving objects are agents. This information in then used to generate reactive behaviors (6 in total), which are then combined together in a tested sequence, resulting in agents progressing through the map with minimal collisions.

Pelechano [7]presented an implementation of Microscopic crowds.This system, called HiDAC (High Density Autonomous Crowds),can simulate large, dense crowds of autonomous agents in confined areas (offices). In this model, agents are entirely autonomous and can be modified to express varying personality traits. Agent behaviors are computed on two levels, a high level module and a low level module. The high level module is responsible for navigation, communication and decision making. The low level module handles perception and motion. These two modules cooperate to simulate the complex behaviors exhibited by pedestrians. This model eliminates unwanted behaviors exhibited by a Helbing [1] based system (agent vibration) and it displays complex social behaviors, such as natural bi-directional flow, queuing behaviors, pushing behaviors, falling agents becoming obstacles and panic

propagation. This system is capable of simulating 600 agents in large indoor areas.

Van den Berg at al [8]presented an implementation of Microscopic crowds in both indoor and outdoor environments. This system is compromised of two levels, a high and low level for simulating human spatial navigation. A pre-computed road-map is used for global path planning and a reciprocal velocity obstacle module for local navigation and collision avoidance. Agents are able to avoid both static and dynamic objects, including non agents. This model works exceptionally well in narrow, crowded areas and does not exhibit agent vibration, however it does suffer from "reciprocal dance" when two agents meet head on in narrow passages. Agents are also incapable of choosing alternative paths when an area becomes highly congested. This system is inherently parallel in nature, agents require no coordination and are entire self localized. Tests show that the addition of extra cores increases performance in an almost linear fashion, with decreasing performance attributed to cache coherency and memory latency.

2.2.1 Advantages of Microscopic systems

The main advantage of Microscopic agent simulation is that agents are represented as individuals, they are able to navigate locally and can plan paths through complex, structured environments. In these model crowd behavior is emergent, so realistic crowd flows and other behaviors can be modeled without loss of agent definition.

2.2.2 Disadvantages of Microscopic systems

The main advantage of Microscopic agent simulation is the complexity of the system, agents are self contained and CPU intensive, making it expensive to simulate large, high density crowds. In reactive systems, agents can suffer from vibration behaviors, caused by repulsion forces, which are undesirable.

2.3 Chosen Simulation Model

For this dissertation, a Microscopic system has been chosen. It allows fine control of actions, while still simulating high level emergent behaviors of crowds. Macroscopic systems often demonstrate homogenous behaviors, which are undesirable and do not demonstrate the varying characteristics of individual agents. A properly tuned Microscopic system can simulate large numbers of autonomous agents in highly complex environments with no loss of agent control, this is perfectly suited to large gaming environments.

2.4 Microscopic model breakdown

As should be apparent, Microscopic systems seem to share certain characteristics. In each of the Microscopic systems presented, agents and the world they inhabit have been abstracted into modules in similar fashions. In each of the papers presented, each of the abstracted modules is implemented in a different way, but with similar results. By reviewing these systems, agents can be broken down into 4 separate intercommunicating modules. They are world representation, path planning, nearest neighbors and object avoidance

2.4.1 Navigation

Most of the previously discussed works [5] [6] [7] [8] contain navigation and path finding modules. Only Pelechano [7] does not go into detail, while the others give a brief overview. These modules are dependant on the world representation. Robust path finding in dynamic environments is complex and extensive research has been put into this field for over 20 years, resulting in vary techniques, each with their pros and cons. A* is considered to be the defacto path finding algorithm for games, both Millington [9] and Buckland [10] have extensive chapters on A*, including varying implementations specialized for games. Games up to this point have been generally static, with small numbers of agents and dynamic objects. A* may no longer be a viable solution for path finding in these dynamic worlds, so extensive research has been put into differing implementations as well as extensions of A*.

Stentz [11] presented a paper on Optimal and Efficient Path Planning for partially known environments, or D* for short. D* is a path planning algorithm for partially known environments in which the state of the world can change. A* was designed to handle static environments. If new information about the environment is discovered, A* must recalculate the entire path from current location to destination. Though the path returned is optimal, this brute force technique is inefficient. The D* algorithm incorporates dynamic changes to the search space in an efficient and optimal manner by recalculating less than the entire path in response to discovery of new information. The method presented in this paper uses 2-dimensional uniform grid to represent the world. Like A*, this algorithm has large memory requirements that grows with the size of the search space. Stentz further improved this model by introducing focussed D* [12]. A heuristic focussing function is added to D*, that focusses the repairs to significantly reduce the time required for initial path calculation and replanning, vastly increasing the performance of D*. Likhachev [13] presented a paper on D*Lite, a new approach to path finding in dynamic environments. It returns the same, optimal paths as focussed D* but it is algorithmically different, bearing a strong resemblance to A*. Results indicate that it is more efficient than traditional D*. Ferguson [14] proposed a multi-resolution D* algorithm that is capable of generating paths across large scale environments, using quadtrees to merge smaller areas of constant cost into larger areas. This results in a much smaller search space, provided that areas of unform cost exist.

Gonzalez [15] presented a paper on generating multi-hierarchical graphs, which can be used for faster,

more efficient path finding. Graph searching is an expensive and memory intensive operation, A* for example, stores every possible path during a search. This method combats this by abstracting graphs into multi-level hierarchies. Lower level nodes are grouped into a singular node, which form part a new graph at a higher, abstracted level. The higher the graph level, the fewer nodes there are, until you reach the highest level where there is only one node. When performing a search, the correct hierarchy level is chosen and searched, this happens recursively down each level until a path is found at the lowest level. This model removes needless path searches and omits unnecessary nodes. Searches on multi-hierarchical graphs retrieve the same paths as low level searches in 98% of cases, while performing significantly faster with reduced memory consumption.

Korf [16] presented a paper on RTA* (Real-Time A*). RTA* is similar to IDA* (Iterative Deepening A*) in that it has a limited search space. This model assumes that finding a complete path in one search is impractical and expensive, so instead the search is limited to a preset horizon, it can only make a certain amount of queries before it must choose a path. RTA* adds an extra heuristic that adapts to gradually improve the performance of searches. This model is useful because resources are limited in game environments and finding full paths is expensive, limiting the depth of searches would greatly decrease overhead, and the guarantee that the best path would be chosen is appealing.

2.4.2 Nearest Neighbor/Spatial divisioning

Most of the previously discussed works [5] [6] [7] [8] require agents to examine their immediate environment. Agents must ascertain which objects and agents should be considered for avoidance. Nearest neighbor queries are expensive, especially in large, complex environments. In order to combat this, spatial divisioning techniques have been researched to significantly increase the speed of this operation.

Ericson's [17] book on Game Physics proposes various systems for spatially sorting objects, giving the pros and cons for each. These include uniforms grids, quadtrees, octrees, loose octrees and k-d trees. Primary focus is placed on BSP-trees, due to their versatility and ease of use.

Samet [18] presented a paper giving an overview of hierarchical data structures that can be used for spatial divisioning. The simplest of which is a quadtree, it is a 2-dimensional divisioning technique that recursively splits the environment into quadrants, until each quadrant contains less than n objects. Octrees extend this model by adding an extra dimension, so that each quadrant is now a 3-dimensional cube. Two extensions of the quadtree model exist, the PR Quadtree (for point data) and the PM Quadtree (for polygons). Quadtrees/Octrees are inexpensive to generate and are widely used to represent spatial data. Shao [6] uses quadtrees to store perception and paths maps. To date there have been no documented cases of quadtree generation on a GPU using CUDA.

Guttman [19] introduced the R-tree. The R-tree is a hierarchical data structure used to store rect-

angular objects in close proximity. R-trees can be useful in physics systems that rely on AABB's for object pruning. Beckmann [20] presented a paper introducing the R*-tree. The R*-tree extends the existing R-tree model and it greatly outperforms already existing R-tree techniques such as Greene's R-tree, quadratic R-tree and the linear R-tree. It is also capable of handling nonuniform data distributions, which can occur in physics engines. De Berg at all [21] further developed the R-tree, producing the Priority R-tree. This model is worst case optimal and is able to answer any window query in linear time. Results indicate it significantly outperforms existing methods when dealing with extreme shapes and distributions.

G. Luque at al [22] presented a paper on Semi-Adjusting BSP trees. This paper extends the already existing BSP (Binary Space Partitioning) model often used in physics engines and other systems that require Broad-Phase Collision Detection. BSP trees are simple to implement but must be re-evaluated every time the environment changes (ie. when there are moving objects). The Semi-Adjusting BSP trees fixes this problem by re-balancing nodes of the BSP tree only when necessary, so this system never requires a rebuild of the structure, which greatly improves performance. Semi-Adjusting BSP trees show promising results in both sparse and highly cluttered environments, it is able to handle thousands of objects in real-time and shows better performance than Loose-Octrees and Spatial Hashing. There are negligible performance differences when compared to Quad-trees.

Garcia [23] presented a paper on Fast k Nearest Neighbor Search using GPU. Rather than building Data structures for nearest neighbor searches, it may be possible to brute force the problem using the GPU. This CUDA based system improves the kNN search by up to a factor of 400 compared to a brute force CPU-based implementation. It also returns a set number of objects that are guaranteed to be the closest to the agent, in previously discussed Object Avoidance modules only a limited number of obstacles can be processed at once, giving more merit to this system.

Green [24] published a whitepaper on the implementation of a particle system in CUDA. This system was able to simulate over 40,000 particles at 120fps. This system is one of the first to implement uniform grid generation on a GPU using CUDA. The key to his algorithm is a GPU efficient version of radix sort, developed by Nividia [25], which is used to sort particles into their appropriate grids. Particles then query their surrounding cells for possible collisions with neighboring particles. This system is able to simulate over 40'000 particles at 90 fps on a GTS 8600.

2.4.3 Object Avoidance

Most of the previously discussed works [5] [6] [7] [8] offer different equations and techniques for obstacle avoidance, each allowing agents to avoid obstacles and other agents, so that movement is free flowing and natural. This problem has also been extensively researched and there are many varying methods

proposed to solve it, including techniques for vehicle and robot navigation.

Sebastien at al [26] proposed a novel method for solving interactions between pedestrians and avoiding inter-collisions. Rather than use a simple repulsive force, agents scan their environment and choose a new velocity that will maximize speed and minimize collisions. This is done via the sampling of every agent/obstacle in range and calculating their future positions at discrete time-intervals. Two new velocities are calculated, one to overtake the obstacle and one to let it overtake them. The best possible solution is taken from these samplings and is applied, resulting in realistic, collision free movement. Real world data comparisons suggest that this method works best in low density areas. This model can only simulate 150 agents in real-time because of high CPU overhead per agent.

Yang et al [27] proposed a system for vehicle collision avoidance. It is a multi-agent model that decomposes the world into a set of agents, each with their own mass, position and velocity. Agents come in 2 forms, obstacles and decision agents. Interaction behaviors are modeled as the inverse of Newtons Law of gravitation between these objects. Agents repulse each other depending on their proximity. An agents perception range is larger for static obstacles than it is for other decision agents, since it is assumed that agents will try to avoid each other while an agent is itself entirely responsible for avoiding an obstacle. This model assumes that agent interaction (collision) is catastrophic and is designed to avoid this at all costs. Result indicate that their solution is adaptable, flexible and reliable but further work is needed for adequate moving obstacle avoidance.

Braun et al [28] produced a paper that examines the impact of individual agents characteristics in emergent crowds. They generalize Helbings equations in order to add individualism to the agents. The primary use of this system is to simulate the motion of crowds evacuating an area in complex environments. Group behaviors can be seen is a consequence of these individual parameters. In one scenario, altruist individuals were modeled, instead of immediately saving themselves they tended to save others first. This system shows a 20% improvement in the flow of people when compared to Helbings model. This system suffers from the inherent problems of a physics agent simulator (ie. agent vibration) but it's simplicity means that vast numbers of agents can be modeled.

Reynolds [29] paper outlines how to construct basic motion for autonomous agents in a three level hierarchy of: action selection, steering and locomotion. The paper presents a number of common steering behaviors and their means of implementation. This includes an object avoidance behavior that gives agents the ability to manoeuver in cluttered environments. This behavior enables an agent to avoid only one obstacle at a time and if the environment is too cluttered it is possible for agents to get stuck. Other interesting behaviors include flocking, which can simulate groups of agents staying in formation. This system offers realistic behaviors, including deceleration when in close proximity to an obstacle, through very simple means. It should also be noted that Reynolds paper is considered the defacto basis for agent locomotion in gaming environments [10] [9] [**?**].

2.4.4 World Representation

Most of the previously discussed works [5] [6] [8] abstract their representation of the world in differing ways. Agents are given a simpler, abstracted view of their world that they use for navigation and decision making. There are many ways to represent a digital environment, each with their own pros and cons. This includes representation of the walls, traversable areas and the objects within the environment.

Representing objects as their polygonal mesh for collision avoidance could prove very costly and it is best to generalize these objects and to represent them in simpler formats. Physics engines commonly simplify objects, creating bounding volumes to represent said objects instead of it's polygon mesh. These bounding volumes greatly simplify physic systems, with limited loss of realism. Ericson [17] and Eberly [30] both presented commonly used bounding volumes. AABB's (Axis Aligned Bounding Volumes) are the simplest bounding volumes but they are not rotationally invariant, they must be recalculated whenever an object rotates. OBB's (Orientated Bounding Boxes) are also commonly used, they are rotationally invariant and offer tight fits to complex objects, but intersection tests are more complex. Spheres are the simplest option, they are commonly used by Microscopic systems to represent agents and to generate the resulting repulsive forces. K-dops are another interesting bounding volumes, but they are also not rotationally invariant.

Millingonton [9] discusses the various world representations that are commonly used in games, primarily for navigation purposes. Shao and Terzopoulos [6] use tile graphs to represent the traversable areas of the map, but they are expensive to search and if the tiles are too coarse, agent navigation seems clunky and unrealistic. "Points of visibility" is another popular method for automatic graph generation, but it is incredibly expensive and can result in overly complex navgraphs. Polygons meshes are another useful way to represent a navigation mesh. Vertices can become nodes and the edges of a polygon become graph edges, but this produces poor results, it is often best to treat polygons as nodes and to create connections to other adjoining polygons. This works well when the walkable areas are coarse but the more detailed a polygonal mesh, the more complex the structure to search.

Rabin [31] further recommends the use of polygons meshes but rather than use the coarse mesh offered by the map, a smoothing algorithm is passed over the mesh to merge neighboring polygons, creating larger, convex polygons. These larger polygons offer smoother paths and greatly increase the speed of the path finding algorithm, because the search space is now drastically smaller. Navmeshes are used in Lamarche's [5] Microscopic simulation and offer promising results, their construction is more complex but it removes dead ends and narrow passages from the navmesh. Delauney triangulation is used to create the mesh, this ensures that the mesh is smooth and the polygons are well fitting.

Delauney triangulation is a common algorithm used for navmesh generation and it may be possible to generate fine navmeshes for navigating around dynamic objects in real-time, Majdandzic [32] pre-

sented a paper detailing the use of GPU's to generate Voronoi diagrams, Voronoi graphs being the dual graph of Delaunay triangulation graphs. This technique produces voronoi diagrams as textures and the graph must be extracted from the textures, making them impractical for real-time use. Further research should be put into this field, but for now it is too costly to implement.

Lamarche [33] presented a system titled "Topoplan" that can generate topological maps from unstructured 3D triangular meshes. This system creates a 2D representation of the map that is used by agents for navigation. This system properly identifies obstacles, steps and bottlenecks, while it describes zone connectivity using a small number of waypoints and paths. This system is quite recent and work is currently being put into extending this model.

2.5 State of the Art in games

Kynogon is a middle ware company that specializes in Game AI. Their AI engine, Kynapse, has been used in over 80 games [34], including Battlefield 2: Modern Combat and Crackdown, both best selling games. Kynogon was recently bought by Autodesk, the company responsible for 3D Studio max and various other applications used by games developers.

Kynapse is a highly complex AI engine. It is capable of handling dynamic worlds with complex interactions between agents and obstacles. In a whitepaper released by Kynogon on dynamic path finding [35], they explain that their engine groups collections of dynamic objects together. When an Entities path becomes blocked by an object or a collection of objects, the dynamic pathfinder tries to construct and follow a path that goes around them. Their worlds also include the concept of smart objects, discussed later. These smart objects are tied to the path planner and expose methods to the agent that enable them to navigate around or through them, such as a door or an elevator. Their engine is also heavily multithreaded, resulting in complex game AI that does not tax the CPU.

Another white paper they have released [36] explains how perception can play an important part in game AI but that it is often ignored, resulting in unrealistic behaviors and broken game mechanics. By giving their agents a sense of perception, their behaviors can become more realistic, resulting in deeper user immersion. Demo applications and videos can be found on their website.

2.6 Physics Engines

Physics engine are now considered an integral part of most games. These engines are designed to approximate physics collisions as efficiently as possible, often trading realism for performance, this loss of realism is generally imperceptible and enables these system to simulate large quantities of objects. Physics have become such an important feature that it is now common practice to use an outside engine "Middleware" rather than develop one in-house. Below is a brief overview of both

commercial and open-source engines that are used in modern games.

Bullet [37] is an open source physics engine, it is multi-threaded and offers 3D collision detection, both for soft body and rigid body dynamics. This engine is capable of simulating large numbers of rigid bodies simultaneously. Bullet was the driving force behind GTAIV's physics, which has been applauded for having realistic physics and physics based driving behaviors. The Bullet engine can be downloaded from their website.

Havok Physics [38] is a physics engine developed by Irish company Havok. It was one of the first producers of a middleware physics engine and the company now specializes in physics and physics based animation. Havok is considered to be one of the top physics engines and has been used in over 150 computer games to date, most notably in Half-Life 2, which is renowned for its use of physics. In 2007 Havok gave interested parties access to most of the C/C++ source-code, giving them the freedom to customize the engine's features as they see fit.

PhysX [39] is a physics engine that has been released by the Nvidia coroporation. Originally developed by Ageia, it was intended to run on a specialized Physics card, but this turned out to be impractical. In February 2008, Ageia was acquired by Nvidia and they started work on porting this engine over to the GPU using CUDA. In August 2008 Nvidia released PhysX and offered the SDK free to download and use. PhysX is currently the only physics engine that runs of the GPU. By using the GPU, PhysX is capable of simulating vast number of both rigid and soft body objects at great speeds.

2.7 Analysis

From the past section it can be seen that there are a multitude of ways to simulate large crowds in dynamic environments. From analysis of both Macroscopic and Microscopic simulations, it is clear that a Microscopic system is best suited to the needs of this dissertation. While all the systems mentioned share common traits, they each offer differing implementations. Choosing the correct implementation for each module is of paramount importance. Each module must not only perform its task to specification, but it must also integrate seamlessly with other modules to form a cohesive system.

Chapter 3

Design

This section deals with the design of the intended system. This design must be capable of simulating smooth interactions between large numbers of agents and objects in a complex, physics based world while in real-time.

3.1 Requirements

- Must be able to handle large numbers of agents (500+)
- Must be able to handle large numbers of dynamic objects (500+)
- Must use an existing physics engine, to prove feasibility
- Must run in real time at 30fps
- Must be able to handle large maps.
- Agent collisions (objects/other agents) must be kept to a minimum
- Agent flow and navigation must seem natural
- Solution must be generic - capable of handling varying maps types, eg. Indoor, outdoor, urban, rural.

3.2 Chosen Modules

In the previous chapter a brief overview was given of varying techniques used to simulate microscopic crowds. In this section these techniques will be analyzed and the most optimal configuration of modules will be chosen. These modules must be able to work in tandem and as self contained units. Particular weight will be added to systems that utilize CUDA, as this system aims to take advantage of the parallelism of GPU's and the performance increases they can offer.

3.2.1 Path Finding

After consideration of all the techniques presented, A* has been chosen for use in this system. D* seems like the optimal choice for path finding in a dynamic world, such as the one this project aims to simulate, but upon close inspection it seems that it is ill suited. D* is an agent based path finder, were agents are not omniscient and do not have a full understanding of their world when planning paths. In games, this is not the case, as agents have direct access to all information about their environment. The only beneficial use of D* would be the efficient replanning of paths when the current path becomes blocked, but this is a fringe case. As this is the case, D* and it's varying implementations are considered too costly and complex for this system. RTA* (Real-Time A*) was also considered, RTA* is efficient, has a guaranteed runtime and uses minimal memory. The one disadvantage of this system is that it cannot guarantee that an agent will reach its goal via the current path, it will gradually edge the agent closer to its goal and may eventually find that no path exists along this route. An example would be a search thats heuristic takes it closer to a node, until it realizes that an impassible wall exists between it and the goal node, rendering search pointless. As this system must guarantee that an agent will find the correct path, RTA* cannot be used. While A* has been chosen as the optimal solution, it is still with it's draw backs, such as the branching factor of A* and the expense of finding large paths. This can become incredibly problematic when an agent requests a path that does not exist, for example, in a standard implementation of A*, the search would branch out over the entire graph before it realized that no path exists. This cannot be allowed to happen as it will greatly impede the performance of this system. Hierarchal Path-planning A* (HPA*) could be used to abstract the navigation mesh further, but the maps implemented in this system, while large, will not finely grained enough to warrant its use, instead bas A* will be used and HPA* will instead be considered for future work.

3.2.2 Nearest neighbour/Spatial Divisioning

It has already been stated that spatial divisioning and Nearest neighbor queries are expensive. This system will attempt to use to use CUDA to speed up this process.
R-trees are usefull for storing objects such as AABB's, but these shapes have already been considered invalid, so R-trees are no longer a valid option. Quadtrees and Octrees are simple data structures that are efficient for storing objects, but as yet there has been no implementation of these structures via CUDA. BSP trees are used to split up large number of objects into separate groups, but these groups are not guaranteed to be within a certain distance or range. To date, BSP trees do not currently have a CUDA based implementation. Uniform grid generation could used in a similar fashion to the particle system, but it was designed for sorting objects numbers of over 30'000, far above the needs of this project.
It has been decided that a modified version of the k Nearest Neighbor Search on the GPU will be used. While brute forcing these checks has always been considered an issue on CPU's, GPU's have no such qualms and are well suited to the task.

3.2.3 World Representation

World representation is one of the most important aspects of this system, it describes the way in which the world is represented and interpreted by agents. The traversable areas of the world shall be represented by a navigation mesh. While a tile mesh, as used by Shao [6], is a simple and convenient way to represent the world, it greatly increases search complexity, as well as taking up large amounts of memory in the system. Navigation meshes have been chosen because of the simple way in which rooms can be presented. Each face of a navigation mesh is convex, convexity is important as it guarantees that an agent can traverse from any point to any other point on that face. Agents can also traverse from one face to another is they share an edge. Tile graphs and other similar representations have difficulty with large open areas, while a nav-mesh considers this problem to be trivial. Nav meshes are also suited to indoor areas such as hallways and rooms, which are frequent in game worlds. Navigation meshes are the optimal choice for representing the navigable areas of a game environment because they are generic and can be used to represent any game world in an efficient manner. Representing physics objects and agents is an important aspect of this system. As these objects are expected to be dynamic, the chosen model must be inexpensive and capable of handling moving objects. K-dops and AABB's cannot be used because they lack rotational invariance, every time an objects rotates, it's bounding volume will have to be recalculated, which is an overly expensive operation that should be avoided.
As this is the case, it has been decided that objects will be represented by simple proxy shapes, such as OBBs, spheres and cylinder. These shapes can be used to represent almost any object and will be handled by the chosen physics engine, PhysX.

3.2.4 Object avoidance

This part of the system is incredibly important, it is the means by which an objects avoids obstacles in its path and great are must be taken to ensure it is implemented correctly. Each of the papers referenced features a different technique and set of equations for solving this problem, this would not be an issue except that each of the papers claims that their technique returns the most realistic results.
As it is so difficult to tell which would be the most useful, it has been decided that a prototype environment will created to test each of the equations presented. These technique will also all be combined in different ways, to see if an amalgamated approach will meet with more success.

3.2.5 Overall Framework

This system will be created in a modular fashion so that each compartment can be tested separately, prior to its integration into the main system. This ensures that the system is not completely reliant on one module and will still operate, albeit to a lesser degree.

Chapter 4

Implementation

This chapter will detail the implementation process of this project. This will include the techniques used, the problems that arose and a description of their solutions.

4.1 Framework

As stated previously, this system is comprised of four interacting modules, each representing a different aspect of the system. A framework was created whereby each module is independent and self contained. This enables a module to be implemented and tested independently, prior to its integration into the system. Modules communicate through static methods, whereby a module exposes information to other modules whenever they require it. These modules are as follow.

- Abstraction of the World
- Agents
- Navigation
- Object Avoidance

This system has evolved from the previous chapters proposal and the resulting Framework can be seen in figure 5.1.

4.2 Abstraction of the World

As this is a microscopic system, agents are the predominant feature and they will base their decisions on the current state of the world they inhabit. Despite this, agents do not need to have a complete understanding of their world, they only require as little information as is necessary to make the correct decision. To this end, the world they inhabit is abstracted into simpler, easier to interpret structures. As agents,objects and walls are presented as shapes in 3D space, it has been decided that the physics engine should handle their representation.

System.png

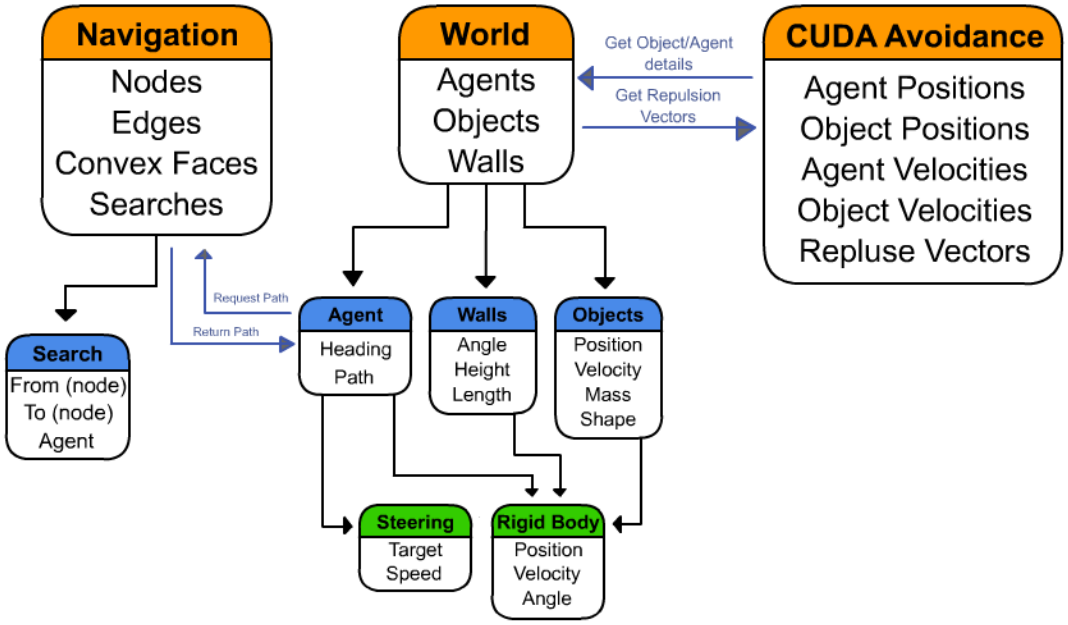

Figure 4.1: The resulting system framework

4.2.1 Physics representation

The chosen Physics engine for this project is Nvidia's PhysX. PhysX is a relatively new technology, so the learning curve is quite steep, especially since physics engines are generally so complex. As this is a physics based project, agents and objects will be represented by rigid bodies. The models in games are generally complex and finely detailed, performing collisions tests against these bodies is incredibly expensive and is generally unnecessary, instead agents and objects are represented by proxy bounding volumes. These bounding volumes are chosen based on the shape of the model, but the most common shape used is that of a simple 3-dimensional box, which is placed around the existing 3D model. Nvidia recommends the use of capsules for agents, because they form the tightest fit. A capsule is a cylinder with a a sphere at either end, they are one of the simplest bounding volumes offered and inter collisions between capsules and other geometry, such as boxes, is incredibly fast. They are suited to agents because of the An example of both these bounding volumes can be seen in figure 4.2.

All rigid PhysX objects share certain properties, they have mass, a position in 3d space and vectors for both linear and angular velocity. These values can be retrieved and modified at any time. PhysX includes a specific character controller class, that can be used to create autonomous agents. However this class is difficult to understand, and the resulting agents are awkward to control, which is a major issue. Instead this project uses a custom made character class, that handles the movement of the agent in the physics based world. It does so by updating an agents position according to its velocity, ensuring that the character is constantly upright by resetting it's orientation matrix. Resetting the

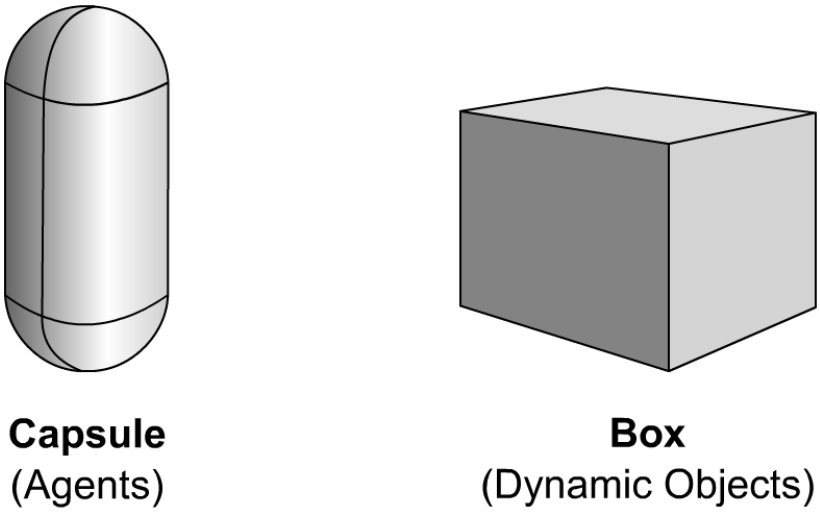

Figure 4.2: The bounding volumes used to represent agents and objects

orientation matrix may not seem to be the optimal solution, but Nvidia's character controller class solves this problem the exact same way. We now have physics based agents that can move about their world, while obeying the underlying physics logic.

4.2.2 PhysX Optimizations

PhysX is incredibly powerful, but as with any physics engine it can be incredibly taxing on the system. So as not to completely cripple the system, PhysX is set to run asynchronously on a different core to the main game loop. The main loop calls the PhysX update function , but it does not wait for this call to complete. PhysX is inherently designed for parallel processing, it uses buffers to store object information, so it is possible to read an object's values, at any time, regardless of wether PhysX is currently updating that object. If too many objects are added then calculations can spill over on the main core, so the number of simulated rigid bodies will be kept to 1500, which is more than enough to prove the concept of this dissertation.

4.3 Agents

Agents in this system are required to navigate from area to area, all the while avoiding other agents and obstacles that are in their path. To facilitate map traversal, agents have been broken down into 2 levels, a high level and a low level. The high level is responsible for goal selection and arbitration, while the lower level handles point to point navigation. The lower level is modeled on Reynolds [29] steering behaviors for agent locomotion, these behaviors are recommended by Buckland [10] as they give agents smooth, realistic movement. This system does not feature a fully developed steering class, as only steering two behaviors are required, seek and arrive. Seek brings an agent towards its target

point at it's maximum velocity, while arrive moves an agent towards a point, applying deceleration forces to slow the agent, causing the agent to stop at its target, rather than over shoot it. The steering class is quite generic and is designed to handle all types of moving entities.

4.3.1 Steering Behaviors

An agents steering layer is comprised of different values, each representing a different aspect of an autonomous agent. Position is the agents current position in space, it is generally represented as a 3-dimensional vector, with coordinates X,Y and Z. Velocity is also a 3D vector that expresses the direction and speed of an agent (it's speed is the magnitude of this vector). When an agent is updated, its position is updated by it's velocity, effectively moving the agent to a new location. So if you wish to change an agents position, you must modify it's velocity. A normalized copy of the velocity vector, called Heading, is also saved, because it represents the directional vector of the agent and is useful in many calculations.
As this is an autonomous agent, it is capable of changing its own velocity by apply force. These forces are self-applied, and are hence limited. MaxAcceleration represents this limitation, it is the maximum change in speed an agent is capable of applying to itself at a single time step. Agents have a MaxSpeed value, which is the maximum speed an agent is capable of traveling, otherwise an agent would gain speed indefinitely. Agents also have a limited turning speed, in this case called MaxTurnRate, which is the maximum change in orientation and agent can have at any single time step. Mass is represented but is not used by the steering class, it is instead stored in the physics engine as stated previously, and it will be used by the avoidance module, which will be discussed later.

```
Steering class properties:
    Mass             float
    Position         vector
    Velocity         vector
    Heading          vector
    Speed            float
    MaxAcceleration  float
    MaxSpeed         float
    MaxTurnRate      float
```

To further understand steering behaviors, please look at figure 4.3. Here can see that an agent wishes to turn and move towards a target point. When doing so it creates a vector from itself to its target, this vector is then normalized and the angle between it and the heading vector is calculated. If the angle is larger than MaxTurnRate, then the angle of the vector is scaled down. Finally the speed of the agent is incremented by MaxAcceleration, unless it is currently equal to the MaxSpeed constraint. The speed is then used to scale the newly generated vector, resulting in a new velocity that is used to update the agents position. Proper tweaking of these values is required but these behaviors can result in agents that turn and behave realistically.

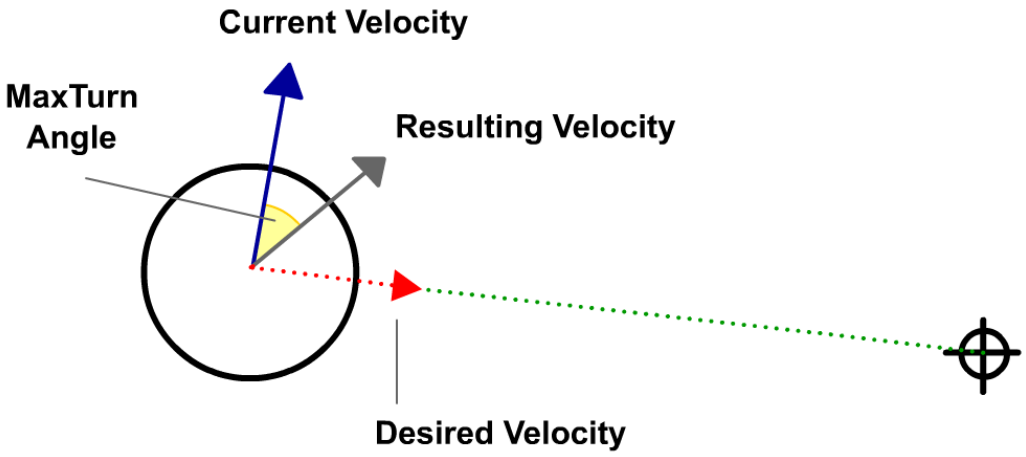

Figure 4.3: An agent turning towards a point as handled by its lower level steering behaviors

4.3.2 Decision making

Agents are required to be autonomous and must be able to make their own decisions based on the world around them. In order to express the individualism of each agents, each agent is given a separate goal from those to their brethren. These goals do not need to be complex, they simply need to demonstrate that each agent is an individual thats fulfilling its own needs. To this end 3 simple higher level goals have been created.

- Follow random path
- Wait/observe
- Wander

At run time, an agent randomly choose one of these 3 goals. Once it has achieved its goal, it then chooses another and continues this process indefinitely. "Follow random path" is a behavior in which an agent chooses two random nodes on the navgraph, finds the path between them and then follows it. Wait/Observe is a behavior in which an agent waits at its current location for a random amount of time, it is possible that an agent will be pushed away from its location, in which case it will traverse back to its original point. Wander is a behavior were an agent wanders around its current area for an indefinite amount of time. These 3 behaviors create the illusion that agents are actively seeking to perform complex goals. These goals can be broken into a combination of very basic sub goals.

- Seek to point
- Arrive at point
- Wait at point

Millington [9], Buckland [10] and Rabin [31] all recommend the use of Finite State Machines (FSM) for goal selection and arbitration. In the above example, a higher level goal is created by combining lower level sub goals. These higher level goals can they be combined together to form more complex goals. This means that new goals can be added to the system with ease.

4.4 Navigation

As is the case in any agent based system, agents must be able to navigate from one area in the map/world to another. The world in its polygonal state is far too complex for agents to interpret, therefore the world must be abstracted in such a way that agents can understand. As stated previously, agents in this system are given the ability to move from their current location towards a 3-dimensional point, provided that their path is not obstructed. By stringing together a series of these points, an agent would be able to navigate from any area to any other area, provided that the transition from point to point is not blocked by an obstacle. Choosing these points at execution time would be incredibly difficult and expensive, it would involve scanning the environment and generating points that lead the agent towards their target, all the while ensuring that agents can navigate from point to point freely. This is essentially an impossible task, so instead, rather than generate these points dynamically, it is best to create them beforehand. To use proper terminology, the points are called nodes and the connections between them are called edges, while the overall interconnected collection is called graph.
With the proper configuration of nodes and edges in a graph, it is possible to create paths across diverse maps with ease. In figure 4.4 we can see an example of a navigation graph. Nodes are highlighted in red, while the edges are coloured blue, if 2 nodes are connected by an edge then it is possible to travel from one node to another unobstructed. Here we see a possible path between nodes "start" and "end" highlighted in green.

Creating these nodes if a difficult task, the worlds created today are far too large and complex for manual node placement, so automatic node placement techniques are generally used. Buckland [10] recommends using a flood fill algorithm that spreads out across the map, creating nodes and edges wherever there is free space. This ensure that it is possible to find collision free paths all across the map, but the resulting navgraph is finely grained (like a tile graph) and is ill suited to the large worlds that games and this project wish to simulate. Other techniques include a line of sight graph, where nodes are placed automatically and are connected via a line of sight algorithm. Often this results in overly complex graphs [9], especially in large, open areas. Another problem with these techniques is that the agents have a very poor understanding of their world, they're only aware of the very distinct edges they can traverse, they have no concept of walls or geometry. It is because of these issues that this project opts to use navigation meshes.

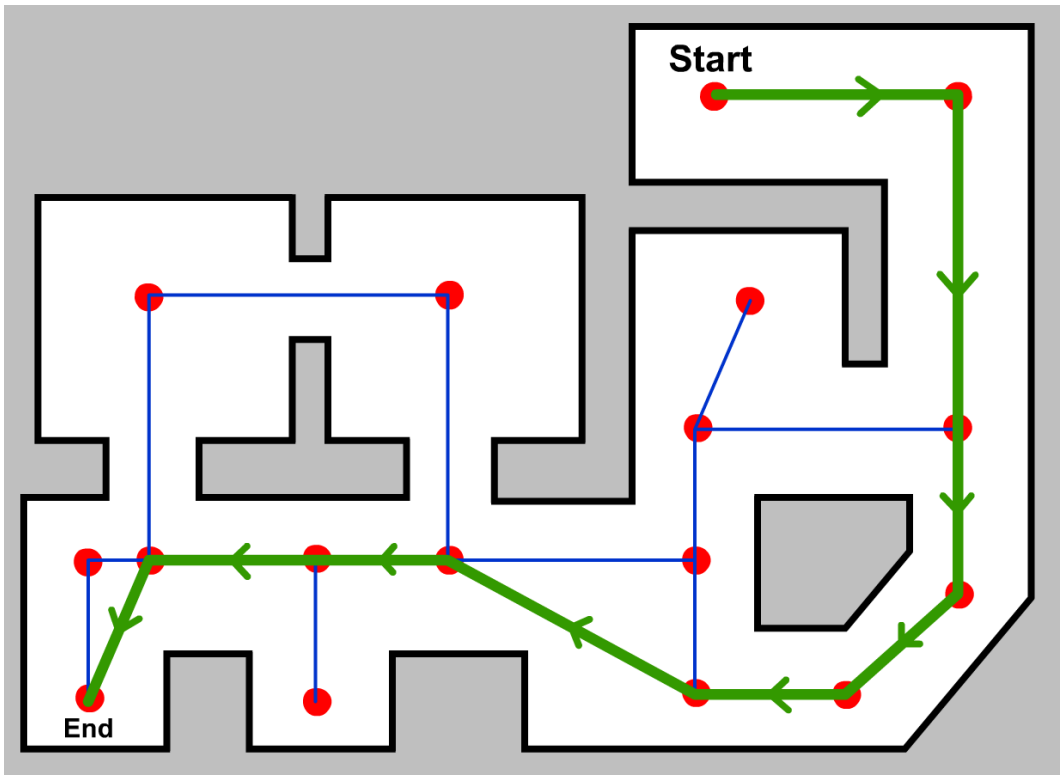

Figure 4.4: An example of a navigation graph with nodes and edges

4.4.1 Navigation Meshes

Navigation meshes (nav-meshes) are considered to be the most efficient way to automatically generate a navigatable graph [9] [5]. The traversable polygons of a map (ie. the floor) are taken and grouped together into larger, convex polygons/faces. In order for a polygon to be convex, its sides must never bend inward towards the center. Convex faces greatly reduces the complexity of both large and small areas, resulting in simple, abstract shapes. The convexity of these faces is important, it guarantees that an agent can travel in a straight line from any point on the polygon to another other.
After these convex faces have been created, it is now possible to create a navigation graph from the resulting mesh. If 2 convex faces share a side, this means that it is possible to travel from one to the other by traversing that side of the polygons. This means that the convex faces themselves become the nodes, while the shared sides become the edges. This creates a simpler, easy to generate graph that is suited to the abstraction of both simple and complex maps. In figure 4.5 the same map as the one in 4.4 has been converted into a navigation mesh, each convex face is represented by a block of colour, while the shared edges between them are highlighted in white.

When an agent traverses this graph, it immediately travels towards its goal edge in the current face, once it reaches this edge, it is aware that it is inside the next convex face it wishes to traverse and it then repeats the process, until it reaches its target. This leads to smoother, more direct mesh

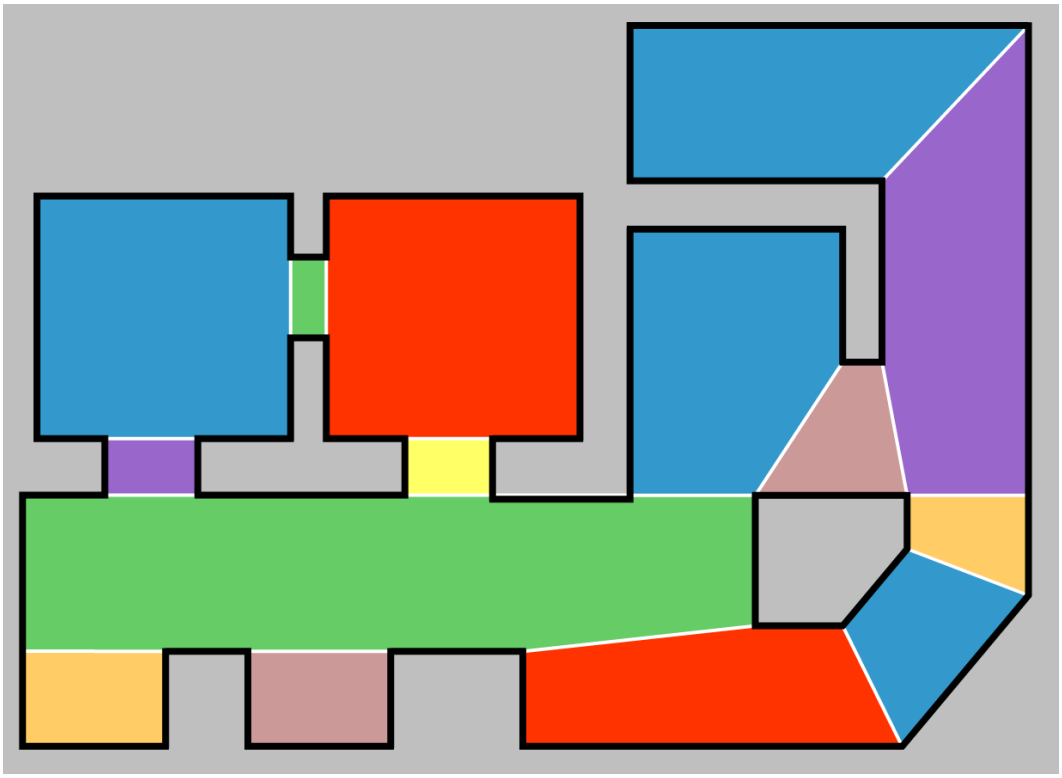

Figure 4.5: An example of a navigation mesh with edges highlighted in white

traversal, while also ensuring that agents do not travel on rails (as can be seen in most games that use generic navigation graphs).

4.4.2 Path Finding Algorithm

While the above navigation mesh is easy to generate, it is useless if the agents are unable to plan paths from area to area. For this reason a path finding module has been created to use the above the mesh. Path finding is the method by which a path from a one to another node is calculated. An important aspect of this behavior is that is must return reasonable paths, meaning that the path returned should be the shortest path out of all possible path. This problem quickly balloons in scale as the size of graph increases, adding nodes and edges greatly increases the number of possible paths that must be searched. In games, the shortest path is literally that, the path with the shortest total length than any other path. To facilitate this, an extra variable is added to the edge class, called cost. This value is aptly named as it represents the cost of traversing that edge, it is generally set to the straight-line, euclidian distance between the nodes of that edge. The total cost of a path can be calculated by summing all edge costs in that path. In games this cost is pre-computed, as nodes and edges tend to be static.

This system relies on A* for finding the shortest path in the most efficient manner possible. A* is

considered to be the defacto path finding algorithm used in games, it is recommended by Millington [9], Buckland [10] and Rabin [31]. A* is based on Dijkstra's algorithm, one of the earliest path finding techniques. Dijkstra algorithm works by spreading out from the start node along its edges from node to node, keeping track of the edges it has traversed thus far. once it has found the goal node, it returns the list of nodes that it followed, returning this as the resulting path. Dijkstra's keeps a list of all the nodes it has looked at thus far, as well as the cost of traveling to that node from the start node. When the search iterates, it chooses the node in the list with the smallest cost so far and adds all its neighbors to the list. This is repeated until the goal node is reached. Dijkstra is expensive and will spread across the graph in unnecessary directions, using excessive amounts of memory. Overall it has a performance complexity of $O(n^2)$, which is is far too costly. This lead to the creating of A*.
A* is modification of Dijkstra that greatly reduces the number of paths checked, resulting in greater efficiency and cost effectiveness. A* works by using a heuristic function that estimates the most effective node to use, in this case the heuristic function is based on euclidian distance. When selecting a node to expand, both its cost and heuristic cost are taken into account, as well as the heuristic cost of that node. This means that a node that is closer to the goal node is far more likely to be chosen than one that is far away. The resulting algorithm is fast, efficient and is guaranteed to return the shortest path from any node to any other node, as long as this path exists.

4.4.3 Path Finding Module

This simulation is required to handle 500+ agents, so it requires a stable, efficient path finding system that will handle the path finding queries of every agent. When an agent requests a path, an instance of a search is created and placed in an ordered queue. Each instance of a search contains all the relevant information for that search, which includes the start and end nodes, the list of possible paths expanded thus far and a pointer to the agent that created it. Whenever a path is found, the calling agent is notified that a path is now waiting for it.

As has been stated previously, path finding is expensive, especially on larger graphs. The simplest method of updating this module is to pop searches of the top of the queue, in a first come, first serve basis. This search would then be iterated through to completion and the resulting path would be returned to its calling agent. At first glance this seems optimal, agents do not need to wait for paths as they are returned almost instantly. After implementing this solution, its problems quickly become apparent. It is impossible to estimate how long it takes to find a path, it could take anything from one iteration for a short path, to $O(n^2)$) iterations int the worst case scenario, where no path exists and the search expands across the entire graph, eating up memory and CPU cycles. This results in a path finding module that behaves erratically, which is a serious issue, as games must have fixed run time so that frame rates are smooth. Another issue is that agents must wait for other paths to be completed before theirs is even considered, even if their path is relatively easy to compute, this is wasteful and should be avoided.
Buckland [10] proposed a solution to both these problems, by limiting the number of iterations a search is able to perform each time step. This way, a search is guaranteed to take the same amount of

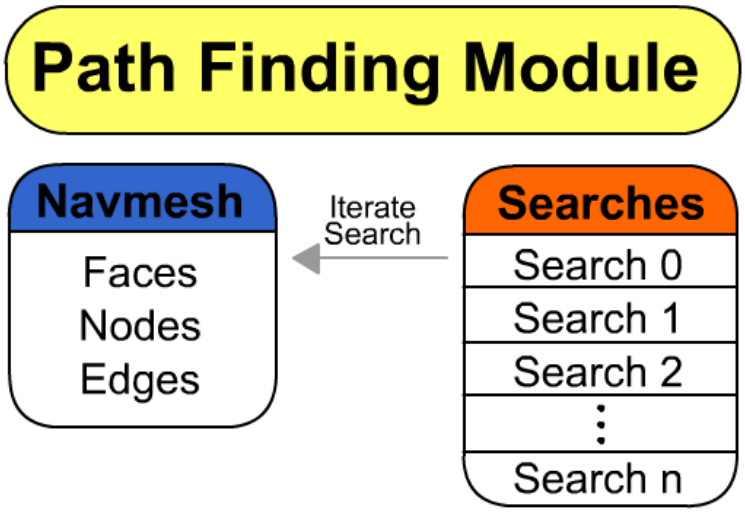

Figure 4.6: The resulting Path Finding Module

time each update cycle, regardless of the search complexity. Agents may have to wait longer for their search, but this will have negligible effect on the how the system looks and behaves. This solution is further developed, by spreading the number of iterations over all searches in the queue. In this method, the number of allowed iterations is divided by the number of active searches, the resulting number is used as the number of iterations each search is allowed to perform. This guarantees that agents are not left waiting for their path while other, longer paths are being calculated. To finish, the path finding module now has a maximum guaranteed run time.

4.5 MapMaker

In order to fully test this system a wide variety of maps must be generated and tested. The original plan was to generate maps procedurally, but the implementation of this was soon found to be far too complex and beyond the scope of this dissertation. Using an existing map editor was also an option, but existing map editors are game specific and documentation for interpreting the generated maps manually was non existent. Based on this information, it was logical step to create a simple map editor that can easily produce varying styles of maps.

The map maker was coded in C-sharp because of the ease in which small application can be made as well as its built in .NET functionality. This system is based of the concept of inter-connected convex faces, as the navigation system uses a navmesh representation of the world.The application gives the user a top down view of the world. The user creates faces one at a time, clicking the map creates a point at the mouse co-ordinates. By clicking multiple times, a series of points are created. The user then signals to the application that it wishes to create a convex face from the points generated,

see figure 4.7 for an example of this. The system ensures that convex faces are created, so that the representation of the world holds true.

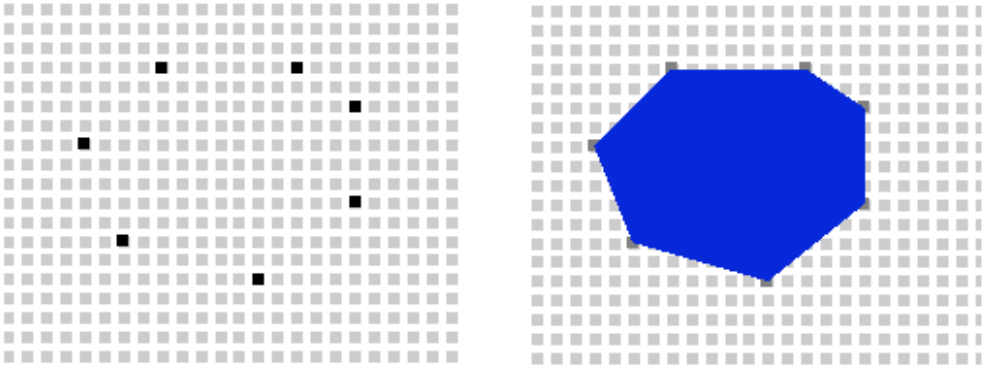

Figure 4.7: A series of points converted to a convex face

Joining faces to creates paths is simple, just create a new convex face that shares two of the nodes of another face. The application automatically realizes that the 2 faces are connected and it creates a link between them.Once the user is finished creating a map, the system automatically generates a navigation mesh, which included the interconnected node-graph used for navigation between meshes. This map is then saved in an easily interpretable file format for the main application. It is important to note that maps can be reopened, modified and saved in this map maker application.

4.5.1 Saving maps

Maps are saved using XML (Extensible Markup Language), it is a textual data format, so the information is human readable when opened in any text editor. XML was chosen because of its simplicity, generality, and usability. Maps in this example are composed of 3 elements, convex faces, nodes and edges. Faces are stored as self contained objects with a list of nested points representing the vertices of said face. Nodes contain a single point in space, which is its location, as well as the radius of the node (used for proximity checks). Edges are saved as simple objects with only two attributes, the id's of the start and end node of this edge.

4.5.2 Loading maps

These maps are loaded into the application using tinyXML, a C++ based open source XML library. It reconstructs the base elements of this map in 3D. Walls are created all the edges of each convex face, unless the edge is shared, in which case it is an opening. The resulting 3D map can be seen in figure 4.8. These maps can be scaled up or down, resulting in both large and small maps, similar to the kind that are found in modern games.

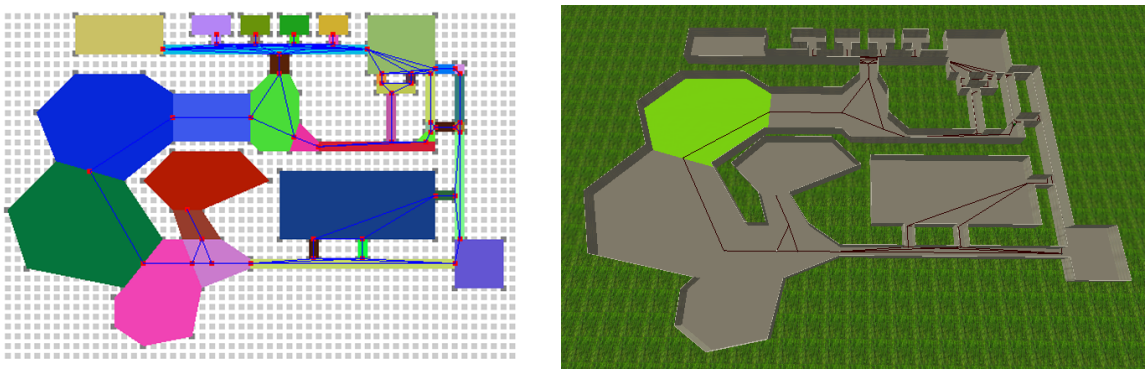

Figure 4.8: 2D map in C-sharp loaded as a 3D physical map, including nodes and edges for navigation.

4.6 Collision Avoidance

As stated previously, agent must be able to navigate through their world without colliding with other agents and objects. This requires agents to scan their environment for their nearest neighbors and objects that may end up blocking their current trajectory. Each of the researched papers presented different equations and techniques for solving this problem, each claiming to result in the most realistic behaviors. Rather than immediately porting and testing all these techniques in CUDA, a prototype environment was created in C-sharp. This environment was used to test each of the techniques presented, in order to ascertain their pros and cons. This project required that agents demonstrate the following characteristics when avoiding agents and obstacles.

- Collision anticipation/ avoidance
- Smooth flowing movements
- Minimal agent oscillation
- Emergent behaviors (lane formation)

After much testing, it was found that agents requires a combination of the techniques presented in each of the referenced papers on Microscopic systems. The implemented technique is explained below.

4.6.1 Perception range

Agents are not required to avoid every obstacle in their world at once, they only need to avoid obstacles that are in their immediate vicinity, specifically obstacles that are in their perception range. The perception range of an agent is modeled after the rectangle of influence, as presented by Pelechano [7] in the HiDAC system.

The rectangle of influence projects out from the agent in the direction that it wishes travel. Any agents or obstacles that are within this rectangle are considered for avoidance. This rectangle is not required to be static, different sizes of agents can be created with varying rectangle dimensions, increasing or decreasing their level of awareness about their world.

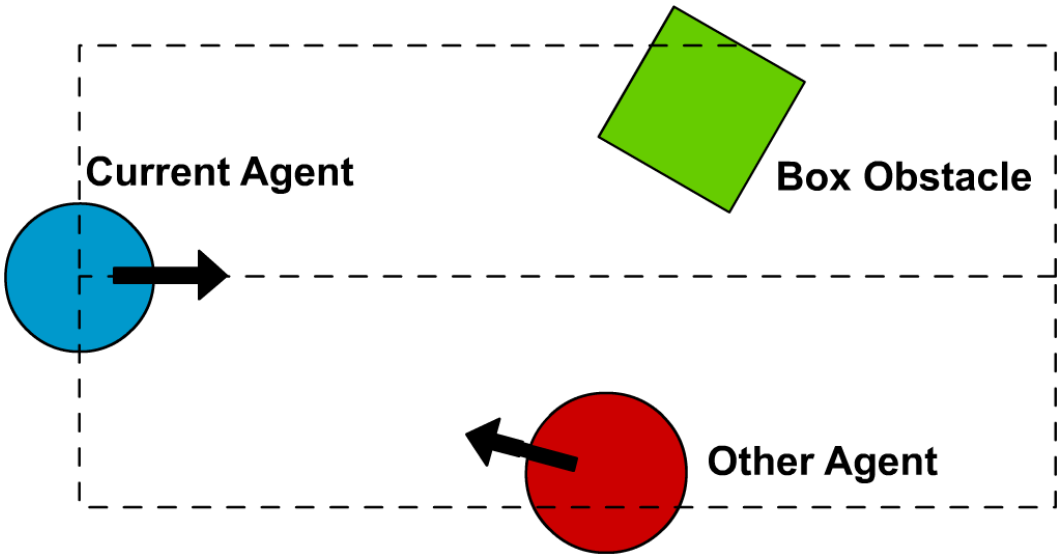

Figure 4.9: Example of an agents perception range, modeled as a rectangle that an agent projects in front of itself.

4.6.2 Avoidance Forces

The equation for collision avoidance is an amalgamation of 2 different techniques presented in differing papers. The majority of these equations are modeled on the HiDAC systems governing, as presented by Pelechano [7], but the techniques by which the repulsion vectors are generated is modeled after Lamarche's [5]method. The reason for these choices will be explained later in this results section.

$$F_i^{To}[n] = F_i^{To}[n-1] + F_i^{At}[n]wi^{At} + \sum_{j \neq (i)} F_{ji}^{Ob}[n]w_i^{Ob} \tag{4.1}$$

The velocity change of agent i (F_i^{To}) ensures that an agent move in its desired direction (F_i^{At}), while avoiding other agents and obstacles ($\sum_{j \neq (i)} F_{ji}^{Ob}$), while trying to keep its previous direction of movement to avoid abrupt changes in its trajectory($F_i^{To}[n-1]$). All these forces are summed together with different weights w_i that are based on an agents preferences, these weights are used to scale the importance of each force on the final direction of movement.

The resulting normalized force vector from the above equations is:

$$f_i^{To} = \frac{F_i^{To}}{|F_i^{To}|} \tag{4.2}$$

This vector is then scaled by the agents current speed, resulting in a new velocity vector that is applied to the agent.

Agent and object avoidance

In order for an agent to avoid an object in its rectangle of influence, it must generate a tangential force (F_{ji}^{Ob}) away from it, this force is generated using an equation presented by Lamarche [5]. The tangential force (t_j generated between agent and i and agent j is calculated thusly.

$$t_j = \frac{\frac{d_{ji}}{|d_{ji}|} + \frac{v_i}{|v_i|}}{|\frac{d_{ji}}{|d_{ji}|} + \frac{v_i}{|v_i|}|} \tag{4.3}$$

In this equations, there are multiple vector normalizations these are required so that the lengths of vectors (d_{ji} and (v_i do not affect the direction of the resulting vector.

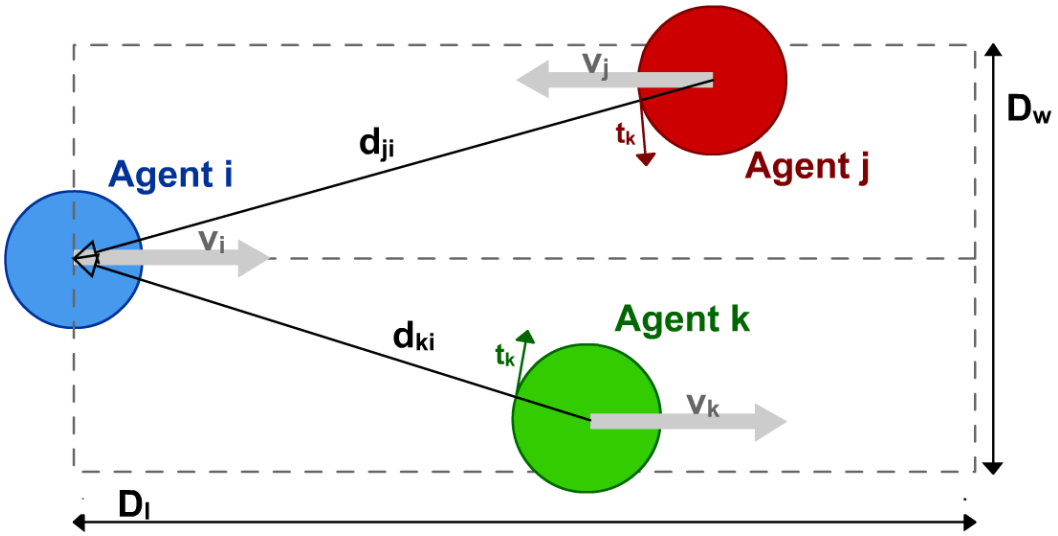

Figure 4.10: An agent generating repulsive forces for 2 agents within its field of influence taking into account their angle of direction and speed.

Next the normalized tangential vector is multiplied by two scalar weights to obtain the final avoidance force.

$$F_{ji}^{Ob} = t_j w_i^d w_i^o m_j \tag{4.4}$$

w_i^d is the weight due to the distance between the agents. It increases as the distance between the agents decreases, thus the repulsive force is stronger the closer i is to agent j, resulting in a more abrupt change. It was found that Pelechano [7] distance weight function increased exponentially as the agents drew closer together, resulting in jarring movements. Instead Lamarche's [5] was chosen, as its weight function grew linearly as the agents became closer, it was also guaranteed to return a number between 0 and 1.

$$w_i^d = \frac{(D_i - d_{ji})}{D_i} \tag{4.5}$$

w_i^o is the weight due to the difference in orientation of the velocity vectors of both agents. It distinguishes whether an agent is moving towards or away from the current agent, if an object is moving away it will generate less of a force than an agent moving toward the agent. This force also applies to moving objects, resulting in agents avoiding both static and dynamic objects in single equation. The below equation is a custom modification of Lamarche's [5] technique.

$$w_i^o = (v_i \bullet -v_j)/2 + 1.5 \tag{4.6}$$

The last weight of the total avoidance force equation is m_j, which is the mass of the agent/object in question. Lamarche's [5] equations do not include this weight, as it assumes that all agents have the same mass. This is not the case in this model, as both large and small objects are taken into account.
These forces are calculated for every agent and then summed together, resulting in a final avoidance force vector that ensures an agent will attempt to avoid all obstacles in its path.

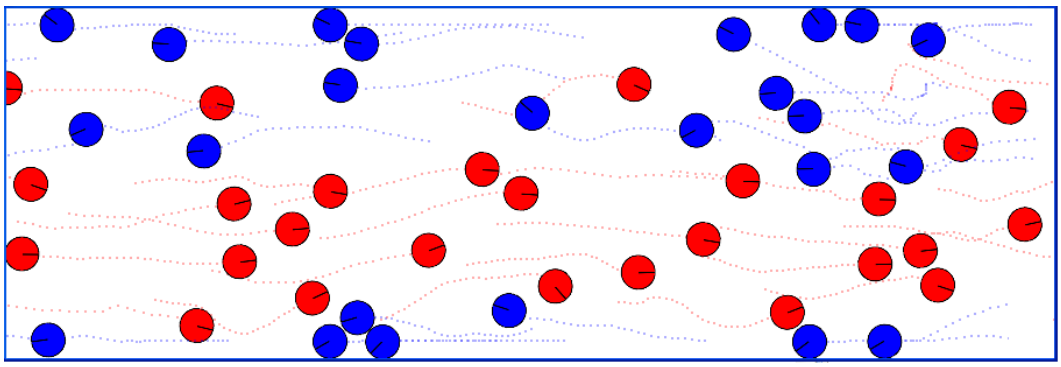

Figure 4.11: Prototype C-sharp application showing the avoidance equations in effect. Please note the avoidance behaviors and the formation of lanes.

The above equations were tested in the C-sharp environment, proving their validity and that the resulting behaviors conform to the characteristics specified at the beginning of this section. Figure 4.11 shows a sample screen from the C-sharp prototype system with the above equations in place. 2 types of agents can be seen in this example, agents whose targets are to the left and agents whose targets are to the right. In this figure the avoidance behaviors can be seen in the trails of agents, another interesting point to note it that agents form lanes, this is an emergent behavior within this system and it is loosely defined.

4.7 CUDA

4.7.1 Overview

Parallel processing on multicore processors is the industrys biggest software challenge

CUDA (Compute Unified Device Architecture) is a parallel computing architecture created by the Nvidia corporation. CUDA is written in a c like language, with certain extensions that give developers access to GPU specific functions and hardware that are unavailable on the CPU. CUDA code compiles via the NVCC compiler, which is an Nvidia specific compiler. CUDA enables developers to write programs that run on the GPU rather than the standard CPU. GPUs are essentially giant parallel processor, a standard 8800 GTS has 128 stream processors, each capable of managing up to 96 concurrent threads, for a maximum of 12,288 threads. Each processors has its own FPUs, registers, and shared local memory. These processors were originally intended purely for graphics operations, such as rasterization, but recent advances in the architecture of these GPUs means that generic user code, rather than shader code, can be run. The advantage of porting code to the GPU is that you can take advantage of its incredibly parallel nature, but only if the problem being solved is itself inherently parallelizable. If implemented correctly, programs can run 65 times faster than their CPU counterparts, as can be seen in [32].

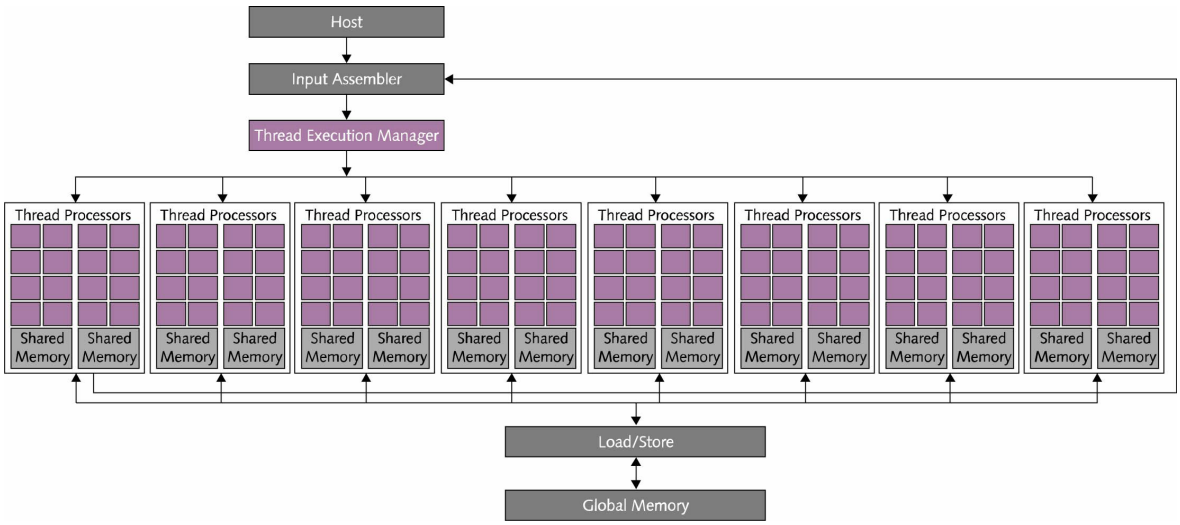

Figure 4.12: Nvidia GeForce 8 graphics-processor architecture.

Of course CUDA is not without its limitations, the stream processors are SIMD, which stands for Single Instruction Multiple Data. SIMD processors work best in unison, so for maximum performance, every SIMD processor should be performing the same instruction as its brethren. This becomes a problem when code relies on branching statements, such as if and else. If you're code is heavily reliant on branching, it will greatly slow down the performance of CUDA, possibly negating the advantage of porting it to CUDA in the first place, so great care must be taken when written CUDA code.

Another issues is that of memory, GPUs do not have caches like regular CPU's, so memory fetches are more expensive than normal. As can be seen in 4.12, memory is off chip, so when a thread tries to access this memory, it stalls, but rather than sit on the processor waiting, the stalled thread enters an inactive queue and is replaced by another thread thats ready to execute. As soon as the stalled threads data becomes available, the thread enters another queue that signals its ready to go. Groups

of threads take turns executing in round-robin fashion, ensuring that each thread gets execution time without delaying other threads.

Threads are run in blocks, blocks are large collections of threads that are all concurrently. Threads in blocks are able to intercommunicate using shared memory, accessing this memory is on chip, so its as fast as accessing local registers. This shared memory can be used to pass information between threads in a block, which alleviates the issue of data dependency between threads.

Thus far, CUDA has primarily been used for high speed computing. The only implementation of CUDA for games is PhysX, Nvidia's proprietary physics engine. This is surprising as Nvidia are trying to put forward the concept that gamers should get 2 graphics cards, one for graphics, and one for CUDA programs. It is this dissertations belief that CUDA can be used to greatly improve gaming technology and performance, and that previously impossible tasks, such as large scale, realistic crowd simulation can be implemented using CUDA.

4.7.2 CUDA avoidance overview

In order to avail of the parallel nature of the CUDA architecture, it was logical to create a thread for every agent in the system, resulting in 500+ threads at once. Each thread would then scan the agents environment, select the obstacles within its rectangle of influence and then calculate the avoidance forces for each one. While this seems simple, proper care must be taken when designing this algorithm. CUDA C is based on C, and C is a very mature language. As such, programmers are used to playing to its strengths, while avoiding its weaknesses. It is common practice for programmers to naively implement their CUDA code as they would C code for a standard CPU application. This should be avoided at all costs, as CUDA has a completely different architecture and thus, a completely different programming paradigm. When porting code to CUDA, programmers must completely reanalyze their implementation in order to make us of this new architecture, resulting in code that would seem strange to a standard C/C++ developer.

Memory allocation and Data transfer

At each time step it is necessary to transfer agent and obstacle data to GPU memory. The attributes required in this process are Position, Velocity and Mass, as well as a vector to store the resulting avoidance force. CUDA is unsuited to data structures referenced by pointers, as it deals poorly with data that is randomly distributed throughout memory, so structs cannot be used. To this end, the position, velocity and mass values are saved as large, 1-dimensional arrays that can be easily indexed into by the calling thread.

Memory creation and allocation is treated differently in CUDA applications. CUDA is poor at dynamically allocating memory at run-time. This can become a problem when new data is added to the system, as is the case in a physics based system when a new object is created. In c or c++ it is recommended that memory for objects is allocated when necessary, so as not the bloat the application with unused memory. This is not the case in CUDA, it is actually much more efficient to allocate memory for the maximum number of objects at program launch. This ensures that adding or removing

objects from the system does not impede its performance in any way. It is important to note that this is standard practice in CUDA applications and it a technique that is highly recommended by Nvidia.

```
cutilSafeCall(cudaMalloc((void**)&dpositions, sizeof(float) * 2000*3));
```

The above sample code allocates an array of 2000 3-dimensional floats that will store the objects positions. It is possible to use the "float3" datatype instead of creating 3 sequential floats, but this is purely semantics and has no affect on performance. The above function is called only once at the beginning of the application, so the memory is constantly available. The total memory allocated for the position, velocity, mass and avoidance variables of 2000 objects is 80'000 bytes, roughly 80kb. Prior to calling the CUDA avoidance code, the CPU copies all object data to the appropriate arrays, and these are then copied over the GPU memory.

Rectangle of influence

As stated previously, only objects that are within an agents field of influence will considered for avoidance. These objects are found by iterating through all objects in the scene and comparing them against the rectangle of influence. This seems like it is a massive waste of processing time, and this would be the case if this application were running on a CPU, where the computational complexity would be in the region of $O(n^2)$. This is not the case in CUDA, it is actually more efficient to brute force this problem than to create a space partitioning structure, especially when the number of agents is below 5000, as has been shown by Vincent et al [23].
When checking if an object is within an agents field of influence, the most efficient technique is to translate that object into the agents local space. This would not be the case in a CPU implementation, as it requires the use of "sin" and "cos" functions, which are CPU intensive. GPU's, however, have dedicated hardware for these functions [41] and a call to either only takes 32 cycles. The "sin" and "cos" values are only computed at the start of a thread and are stored locally for the rest of the algorithm. Once an object has been translated, a simple comparison of an agents X and Y attributes versus the dimensions of the rectangle are required, as is shown in figure 4.13.

Porting the avoidance algorithm

The actual avoidance algorithm itself requires little to no modification. The majority of the calculations used are based on vectors and floats, GPU's have been specifically designed for floating point calculations, so these calculations are performed at little to no cost. This inner pieces of code is performed as soon as an object is found inside an agents rectangle of influence.

Coalesced reading and Code Divergence

As stated previously, CUDA has no cache memory, so accessing memory off chip is expensive. An interesting feature of CUDA is the way in which it handles coalesced memory reads. If multiple threads request the same piece of memory simultaneously, they will all stall, but once the data is returned to

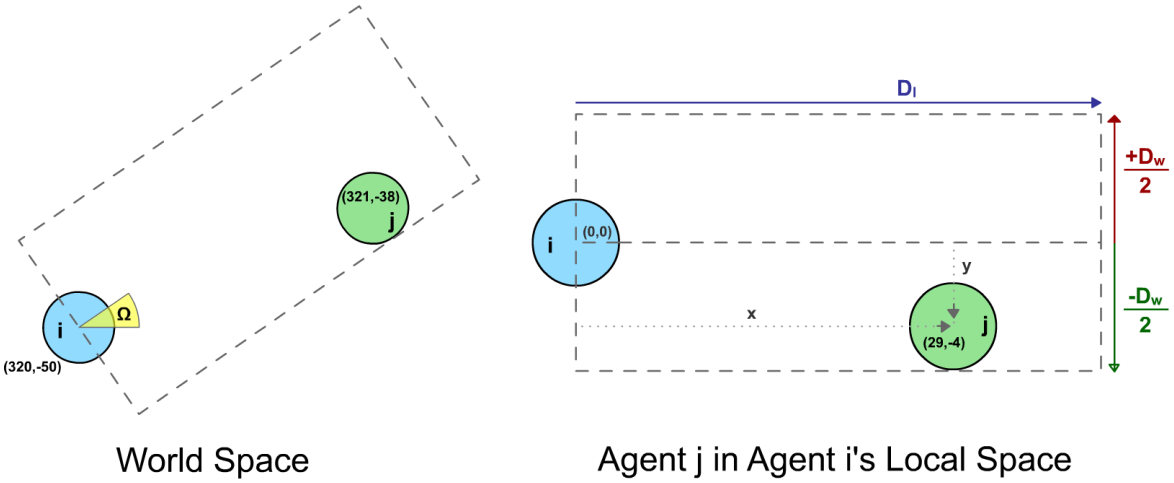

Figure 4.13: Agents j's location in world space, then translated into Agent i's local space for comparison.

one thread, it will be returned to all threads simultaneously [42], greatly increasing the speed of the application.

Firstly, to ensure that there are not multiple, needless memory access calls, multiple registers are allocated per thread to store all necessary information. Secondly, all threads are set to iterate through objects in the same order. Of course, threads will eventually diverge, this occurs when a thread calculates an avoidance force, while all others continue ahead. The more this occurs, the more divergence if found in both code and memory access. A solution had to be found.

```
__syncthreads();
```

To ensure that threads do not diverge, a synchronization function, shown above, is called after every every object is checked. This seems wasteful but the synchronization function is hardware based, so it is incredibly fast and it only synchronizes threads that are in the same block. The feature has the added befit of ensuring that instruction coherency is kept, for the most part, over all SIMD processors, ensuring maximum efficiency.

NaN generation and propagation

Of course, no implementation is without its problems. When testing this application, it was quickly found that it would crash sporadically. These crashes were most frequent at the launch of the application. Whilst debugging it was found that it was not CUDA that was causing the crash, but PhysX, apparently it was being fed invalid values. But the application ran fine when the CUDA module was disabled. It was discovered that CUDA was returning NaN's, and this was causing PhysX to crash. NaN's stand for "Not a Number" and they occur whenever a number is multiplied by infinity, or divided by zero. Whenever a NaN is used in a calculation, the result will also always be a NaN, causing one NaN to become many NaN's, quickly spreading throughout the system.

The distance variable was the cause of this, it is used to normalize the directional vector from an agent to an object, and if the distance is found to be zero, dividing the vector by it results in a NaN. It is possible, at the start of the simulation, that two interpenetrating objects can share the same position, because the physics engine has not yet had a chance to apply a separation force to these objects. The simplest fix was to add an if statement, but this would resulting in code divergence, which was to be avoided. It was found that it was simpler to add a small number, 0.001f, to the distance variable prior to normalization. This ensured that NaN's would no longer appear. The number is so small that is has no discernable affect on the resulting avoidance forces.

CUDA asynchronicity

The CUDA module of the project is asynchronous to the rest of the system, much like Physx. The main game loop calls an update function on the CUDA avoidance module, this function runs on the GPU and the CPU does not wait for it to return. A buffer for the avoidance is used, the CPU reads avoidance forces from an array, which is switched for the latest version whenever the CUDA call finishes. This results in some of the agents being updated by an avoidance force one frame out of sync, this is not noticeable and the asynchronicity of the calls ensure that system runs at maximum speed.

4.8 Rendering

This system will be rendered using OpenGL. It will not feature complex models and lighting because this would take away from the GPU processing power available to CUDA. As stated previously, this project aims to show that games can greatly improved by the addition of a second GPU purely for CUDA specific functions. Sadly the system this project is being tested on has only one GPU, so it would be unwise to limit this GPUs performance with complex graphics.
Despite this, it is still possible to have textured polygons and basic lighting without this taxing the system too severely. Agents are represented as coloured cylinders, while dynamic obstacles are represented by textured boxes. The walls and ground of the world will also be textured.

Chapter 5

Evaluation and Discussion

This chapter outlines how well the system performed. The system is judged in two ways. The first is based purely on performance, the CUDA module is directly compared to a CPU implementation of the same module. The second is based on performance of this system when dealing with differing world maps, in order to ascertain the usefulness of this system in a gaming environment. The test machine for this system has a 2.4ghx processor with 2 gigs of ram and an 8800 GTS with 320 megs of ram.

5.1 GPU versus CPU

To properly understand the speed increases offered by CUDA, the avoidance algorithm was also implemented on the CPU. The CUDA module was then compared to its CPU counterpart, with varying data sets and data set sizes, to ascertain which implementation is best suited to this problem. The CPU implementation has the same algorithm for generating avoidance forces, but uses a spatial divisoning algorithm to speed up rectangle of influence queries.

It was found that the CPU technique outperforms the CUDA technique when the number of agents and objects is quite low. As the numbers increase, the CUDA technique quickly gains ground and becomes more efficient when there are over 200 agents and objects. It should be noted that the CUDA techniques complexity and performance increases almost linearly, due to its multi-threaded nature. The CPU techniques complexity increases exponentially at $O(n^2)$. The CUDA implementation also has a guaranteed maximum run time of O(n), so it is worst case optimal.

5.2 System performance

Three environments were created in order to demonstrate the varying capabilities of this system. Each of the maps are different and are modeled after a specific type of game world.

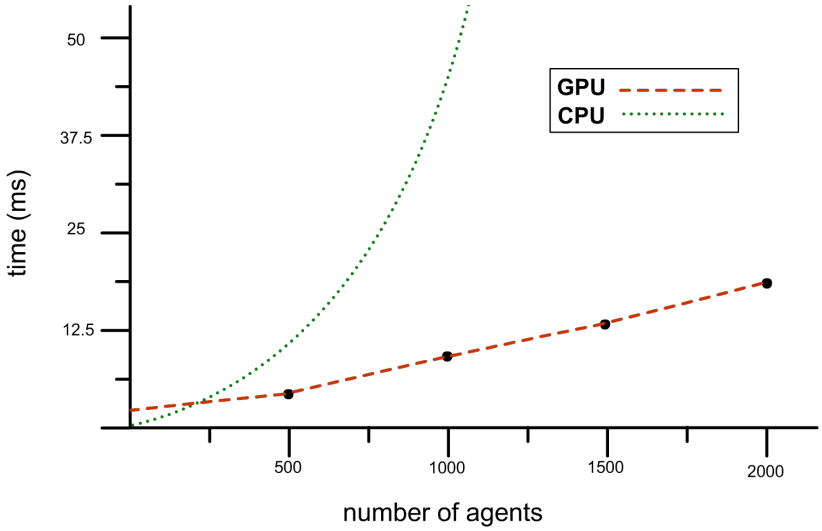

Figure 5.1: GPU and CPU performance comparison

5.2.1 High Density Environment

This test aims to gauge the performance of the collision avoidance behaviors in dense, agent filled environments. A square section of the map is filled with 2000 dynamic agents. The agents are split into 4 groups. Each group crosses the map in one of four directions. In order to test the stability of this system the map is modeled as a torus, this means that whenever an agent leaves the boundaries of the test area, they are transported back to the opposite edge, thus simulating an infinite environment. The colour of agents in the below images are representative of the direction they are traveling.

This test was incredibly successful. Agent collision was kept to a minimum 5.2, while other interesting crowd behaviors, such as lane formation, quickly became apparent 5.3. This system was also incredibly stable and displayed none of the agent vibration issues found in other implementations.

5.2.2 Cluttered Environment

For this test, a large square area is filled with a random assortment of debris, represented by varying sizes of boxes. This is meant to mirror the complex maps found in sandbox games such as GTAIV or Prototype, as these maps are often full of dynamic, physics based objects. 200 agents were created, these agents move towards an area filled with 600 boxes, each set to a random size and position.

Results for this test were found to be very promising. Agents are able navigate across the intersection with minimal collisions 5.4, avoiding agents and obstacles5.5. The agents take an obstacles mass and radius into account, causing agents to avoid larger objects, will stepping over smaller ones. Agents will occasionally bump into these objects, sometimes taking a wrong route. This is to be expected as agents are only given a limited view of their world and have no sense of lookahead or planning. Even with these discrepancies, agents still demonstrate behaviors that have thus far been unseen in games.

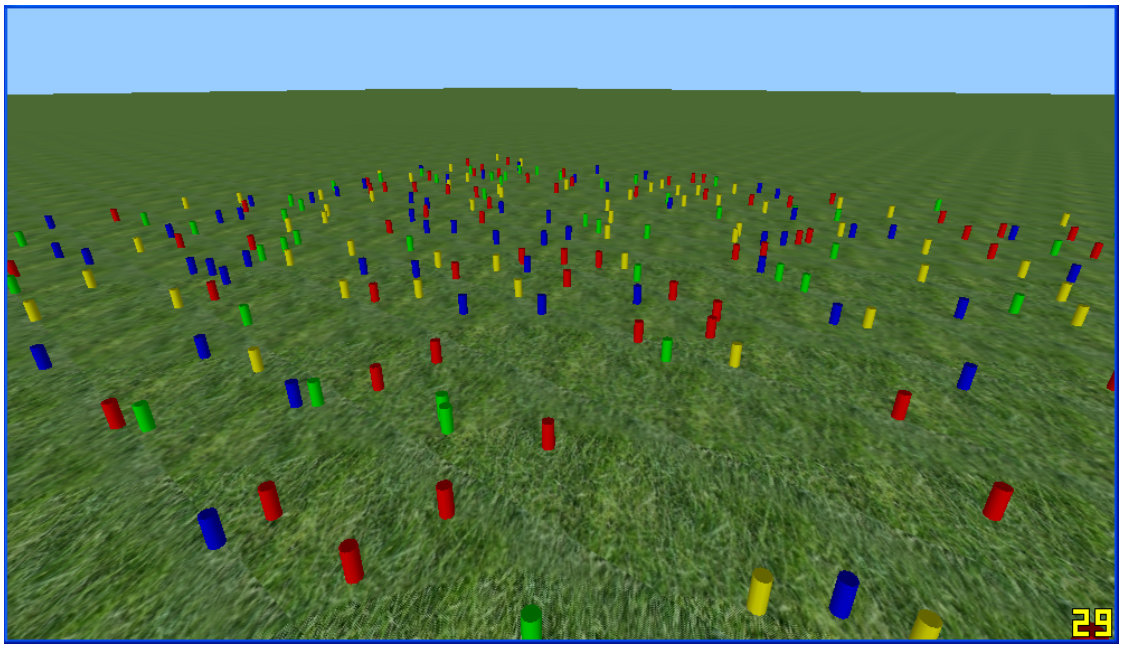

Figure 5.2: 2000 dynamic agents in an enclosed area

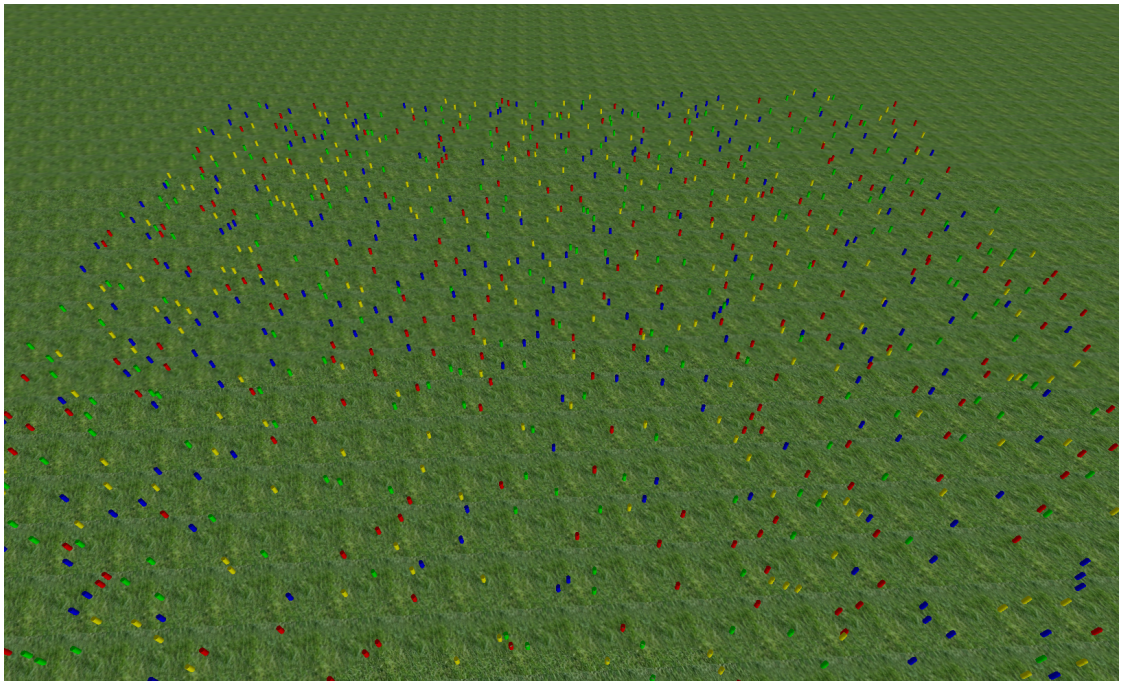

Figure 5.3: 2000 dynamic agents in an enclosed area

5.2.3 Indoor map

For this test a map was created in the accompanying MapMaker, it is modeled after a large industrial complex with large rooms and narrow corridors, these are frequent in games and serve as the ideal

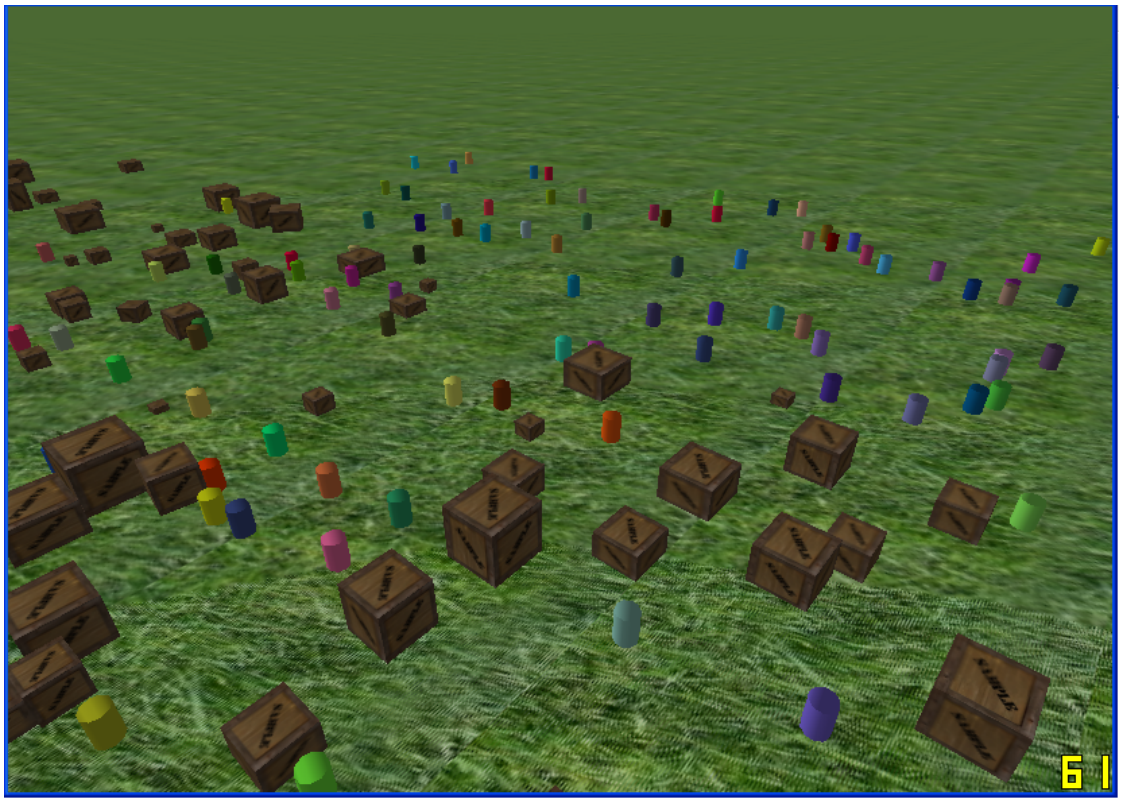

Figure 5.4: 200 agents avoiding 600 randomly shaped boxes

test for the robustness of the pathfinding and navigation mesh aspects of this project, as well as the avoidance behaviors. In this test, 500 agents are placed randomly and given random paths. 500 box obstacles are also placed randomly through the scene. This map will test the full features of the system, putting a strain on each module. This will ascertain the validity of the system and its usefulness in modern games.

This map provided the most interesting results, showing both the advantages and disadvantages of this system. The avoidance behaviors work well in all the areas presented, but jittery movements could been when an agent was in an enclosed space, surrounded by other agents and objects. The path finder was found to be very efficient, capable of handling large numbers of path requests without any one agent waiting too long for its request to be fulfilled. The path finder occasionally chooses congested paths, as it currently has no way of gauging how many agents or objects are contained in a particular face. Even with these shortcomings, this map demonstrates that the implemented system is valid, producing realistic results that are currently unavailable in existing games.

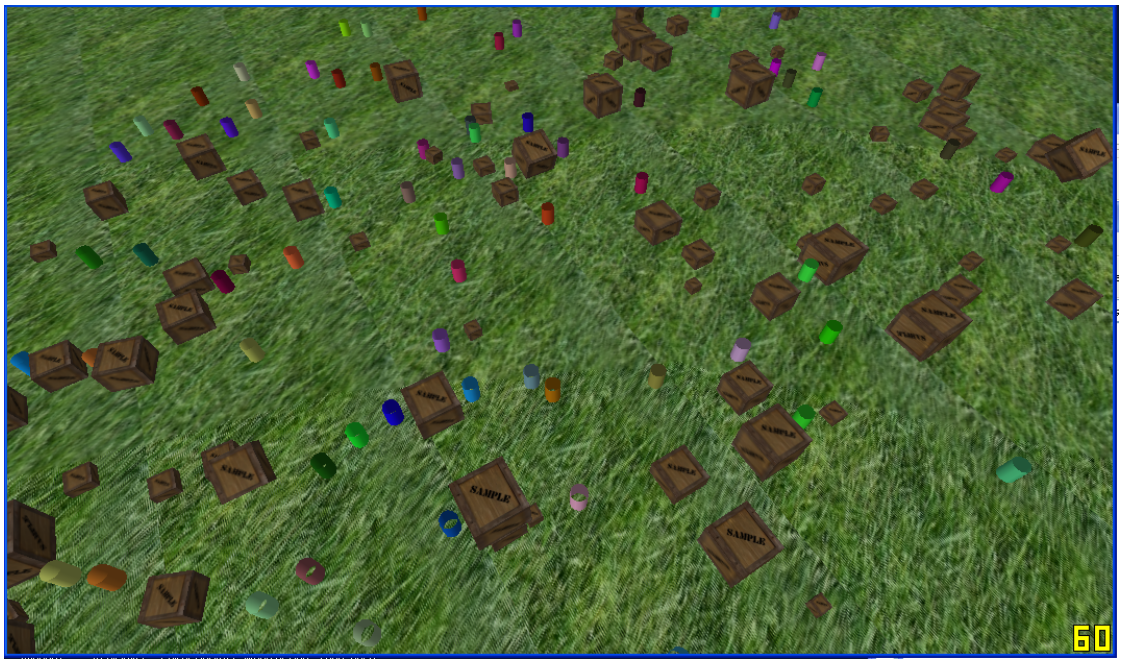

Figure 5.5: 200 agents avoiding 600 randomly shaped boxes

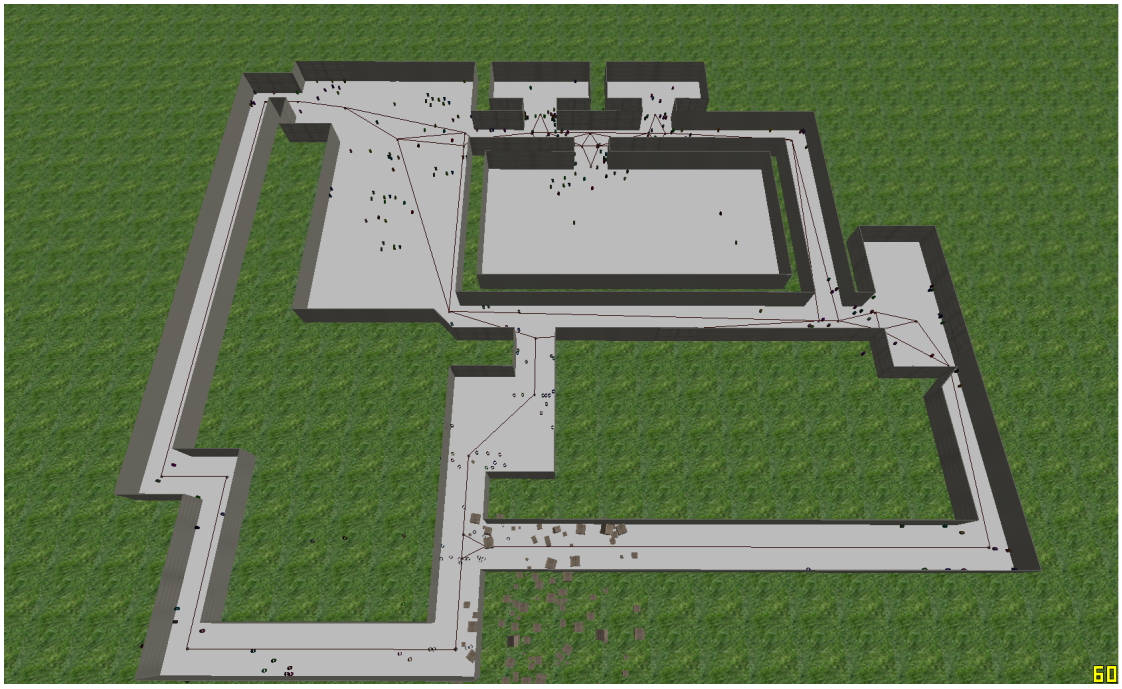

Figure 5.6: Plan view of the industrial map

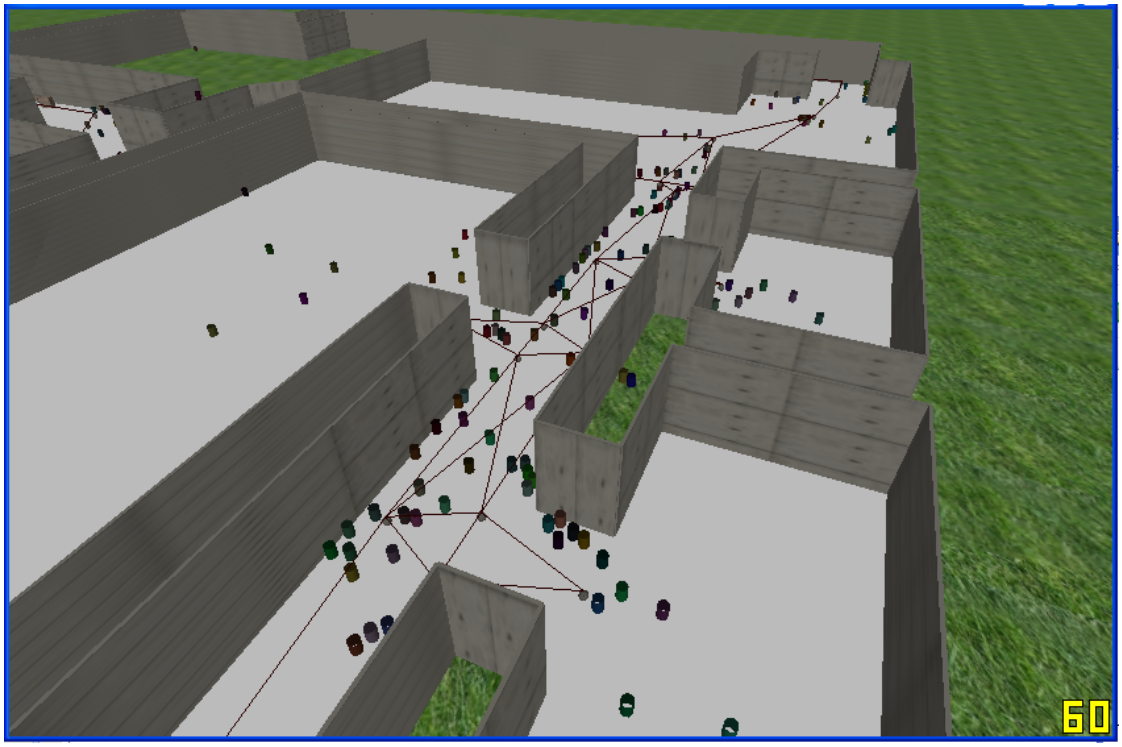

Figure 5.7: Snapshot of a busy corridor

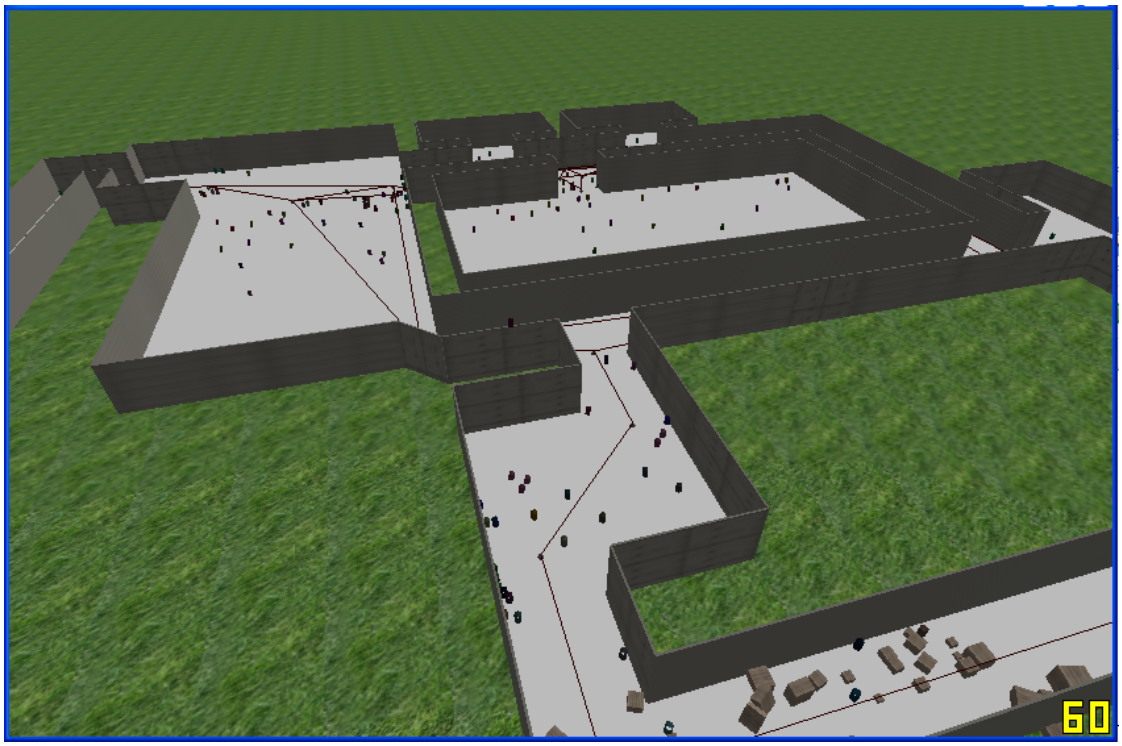

Figure 5.8: Snapshot of agents navigating the map while avoiding obstacles

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation outlined a system that can simulate large quantities of dynamic agents. This system also allows agents to perceive obstacles in their path and to avoid them. These agents display emergent behaviors such as lane formation, while still behaving as individuals with their own goal sets. This system also outlined the method by which an avoidance algorithm can be ported to a GPU using CUDA, making use of GPU specific coding techniques that greatly increase performance. The CUDA function calls are also shown to be asynchronous and that they put little to no strain on the main game loop. This greatly increases the realism and immersion levels of a game at little cost, proving that GPU's can be used in games for more than just rendering.
The system outlined is designed to handle multiple map types, ranging from large outdoor sections to narrow indoor corridors and rooms. The navigation mesh system has been proved to be a robust choice, as it provides better performing navigation graphs with fluid agent movements when following paths. Overall this system is capable of handling almost any type of map, with large numbers of agents and objects, displaying previously unseen dynamic agent avoidance features. This system is inexpensive and could be integrated into an existing game framework, resulting in increased realism and immersion levels.

6.2 Future Work

6.2.1 Congestion avoidance

Currently this system can not tell if an area has become congested and that it should be avoided in future searches. This could be implemented by tallying the number of objects and agents per convexface, this number would then be used to add extra weight the cost of traversing that face, causing the A* search to try less congested paths. Since path costs would then be able to change, it could be beneficial to implement D*, but only for incredibly large maps, like the kind found in MMOs

(Massively Multiplayer Online game).

6.2.2 3rd person testing

At the moment it is impossible to test this system from a player standpoint, as you cannot control a character. It would be interesting to create a character agent that a user could control as if it were a standard 3rd person game. This agent could be given a higher mass than other agents, causing other agents to give the player character a wide berth so that large crowds are not detrimental to gameplay.

6.2.3 Wall avoidance

Currently agents do not avoid static geometry, such as walls, resulting in unrealistic behavior in narrow areas. Giving agents the ability to sense and avoid walls would greatly increase realism, as they would try to keep a comfortable distance from walls rather than walking into them. This could be added to the CUDA avoidance module by iterating through the walls of the convex face the agent currently resides in.

6.2.4 Further convex face simplification

Game frequently feature static pieces of geometry, such as pillars and trees. In a standard path generator, nodes would have be placed around these objects so that paths could be planned around them. This results in extra nodes that increase search complexity. Another approach would be to remove these objects temporarily, create a convex face from the surrounding area and then reintegrate the objects as obstacles that the CUDA avoidance feature would take into account. This would result in simpler navigation meshes and with realistic navigation around these objects, rather than the on rails approach found in most games.

6.2.5 Chaotic scenarios

The scenarios modeled in the previous section are not action orientated and are used mainly as a proof of concept. It would be interesting to see how the existing code would behave in an action oriented environment, such as the kind found in game like Prototype. Avoidance forces need not only apply to physical objects, they could be used to avoid all manners of things like flames, explosion, gun fire and enemy creatures. An other interesting scenario would be to create a zombie simulator This would involve civilians trying avoid zombies, while zombies have a negative avoidance force for civilians, drawing them towards them, causing the infection to spread. The possible scenarios are endless and this technology could be used to implement some interesting game mechanics.

Bibliography

[1] Helbing D, (1992) *A fluid-dynamic model for the movement of pedestrians*, Complex Systems 6 pp 391 - 415

[2] Bauer, D., Seer, S., and Brndle, N. (2007) *Macroscopic pedestrian flow simulation for designing crowd control measures in public transport after special events*, In Proceedings of the 2007 Summer Computer Simulation Conference (San Diego, California, July 16 - 19, 2007). Summer Computer Simulation Conference. Society for Computer Simulation International, San Diego, CA, 1035-1042.

[3] Hughes, R. L. (2002) *A continuum theory for the flow of pedestrians*, Transportation Research Part B 36, 6 (july), 507535.

[4] Treuille, A., Cooper, S., and Popovic', Z. (2006) *Continuum crowds*, In ACM SIGGRAPH 2006 Papers (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. ACM, New York, NY, 1160-1168. DOI= http://doi.acm.org/10.1145/1179352.1142008

[5] Lamarche, F., and Donikian, S. (2004) *Crowd of virtual humans: a new approach for real time navigation in complex and structured environments*, Computer Graphics Forum 23, 3 (Sept.), 509518.

[6] Shao, W. and Terzopoulos, D. (2007) *Autonomous pedestrians*, Graph. Models 69, 5-6 (Sep. 2007), 246-274. DOI= http://dx.doi.org/10.1016/j.gmod.2007.09.001

[7] Pelechano, N., Allbeck, J. M., and Badler, N. I. (2007) *Controlling individual agents in high-density crowd simulation*, In Proceedings of the 2007 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, California, August 02 - 04, 2007). Symposium on Computer Animation. Eurographics Association, Aire-la-Ville, Switzerland, 99-108.

[8] Van den Berg, J., Patil, S., Sewall, J., Manocha, D., and Lin, M. (2008) *Interactive navigation of multiple agents in crowded environments*, In Proceedings of the 2008 Symposium on interactive 3D Graphics and Games (Redwood City, California, February 15 - 17, 2008). I3D '08. ACM, New York, NY, 139-147. DOI= http://doi.acm.org/10.1145/1342250.1342272

[9] Millington, I. (2006) *Artificial Intelligence for Games*, Morgan Kaufmann Publishing, ISBN= 0-12-497782-0

[10] Buckland, M. (2005) *Programming Game AI by Example*, Wordware Publishing, Inc., 2005.

[11] Stentz. A. (1994) *Optimal and efficient path planning for partially known environments*, In Proceedings of the International Conference on Robotics and Automation, volume 4, pages 3310-3317. IEEE, May 1994.

[12] Stentz, A. (1995) *The focussed D* algorithm for real-time replanning*, In: Proc. IJCAI-95, Montreal, Quebec 1995.

[13] Likhachev, M. (2002) *Fast Replanning for Navigation in Unknown Terrain*, IEEE Transactions On Robotics : A Publication Of The IEEE Robotics And Automation Society Volume: 21 Issue: 3 (2005-01-01) ISSN: 1552-3098

[14] Ferguson, M. and Stentz, A. (2006) *Multi-resolution field D**, in Proc. Int. Conf. Intell. Auton. Syst., 2006, pp. 6774.

[15] Fernndez, J. A. and Gonzlez, J. (2002) *Multihierarchical graph search*, IEEE Trans. Pattern Anal. Machine Intell., vol. 24, pp. 103-113, Jan. 2002.

[16] RE Korf. (1987) *Real-time heuristic search: New results*, Proceedings of the AAAI, 1987 - aaai.org

[17] Ericson, C(2004) *Real-Time Collision Detection*, Morgan Kaufmann Publishing. ISBN= 1-55860-732-3

[18] Samet, H., (1988) */An Overview of Quadtrees, Octrees and Related Hierarchical Data Structures*, NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics, Berlin: Springer-Verlag, 1988.

[19] Guttman, A. (1984) *R-trees a dynamic index structure for spatial searching*, Proc ACM SIGMOD Int Conf on Managemnet of Data. pp 47-57, 1984

[20] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990) *The R*-tree: an efficient and robust access method for points and rectangles*, In Proceedings of the 1990 ACM SIGMOD international Conference on Management of Data (Atlantic City, New Jersey, United States, May 23 - 26, 1990). SIGMOD '90. ACM, New York, NY, 322-331. DOI= http://doi.acm.org/10.1145/93597.98741

[21] Arge, L., Berg, M. D., Haverkort, H., and Yi, K. (2008) *The priority R-tree: A practically efficient and worst-case optimal R-tree* ACM Trans. Algorithms 4, 1 (Mar. 2008), 1-30. DOI= http://doi.acm.org/10.1145/1328911.1328920

[22] Luque, R. G., Comba, J. L., and Freitas, C. M. (2005) *Broad-phase collision detection using semi-adjusting BSP-trees*, In Proceedings of the 2005 Symposium on interactive 3D Graphics and Games (Washington, District of Columbia, April 03 - 06, 2005). I3D '05. ACM, New York, NY, 179-186. DOI= http://doi.acm.org/10.1145/1053427.1053457

[23] Vincent, G., Debreuve, E., and Barlaud, N. (2007) *Fast k Nearest Neighbor Search using GPU*, Technical Brief, Nvidia Corporation, June 2007.

[24] Green, S. *CUDA particles*, Whitepaper, Nvidia Corporation, June 2008. Can be found in the CUDA SDK.

[25] Satish N., Harris M., Garland M.(2008) *Designing efficient sorting algorithms for manycore GPUs*, NVIDIA Corporation

[26] Paris, S. Pettre, J. Donikian, S. *Pedestrian Reactive Navigation for Crowd Simulation: a Predictive Approach*, COMPUTER GRAPHICS FORUM 2007, VOL 26; NUMBER 3, pages 665-674

[27] Yang, S., Gechter, F., and Koukam, A. (2008) *Application of Reactive Multi-agent System to Vehicle Collision Avoidance*, In Proceedings of the 2008 20th IEEE international Conference on Tools with Artificial intelligence - Volume 01 (November 03 - 05, 2008). ICTAI. IEEE Computer Society, Washington, DC, 197-204. DOI= http://dx.doi.org/10.1109/ICTAI.2008.134

[28] Braun, A., Musse, SR., de Oliveira, LPL. and Bodmann, BEJ (2003) *Modeling Individual Behaviors in Crowd Simulation*, casa, pp.143, 16th International Conference on Computer Animation and Social Agents (CASA 2003)

[29] Reynolds, C. (1999) *Steering Behaviors for Autonomous Characters*, DOI= 10.1.1.16.8035

[30] David, H. Eberly (2004) *Game Physics*, Amsterdam ; Elsevier/Morgan Kaufmann Publishing, c2004. ISBN= 1-55860-740-4

[31] Rabin, S. (2002) *AI Game Wisdom*, Charles River Media. ISBN= 1-58450-077-8

[32] Majdandzic, I., Trefftz, C., Wolffe, G. (2008) *Computation of Voronoi Diagrams using a Graphics Processing Unit*, In: Proceedings of 2008 IEEE Intl. Conf. on Electro/Information Technology (EIT). IEEE, Los Alamitos (2008)

[33] Lamarche, F. (2009) *TopoPlan: a topological path planner for real time human navigation under floor and ceiling constraints*, The Author(s) Journal compilation 2008 The Eurographics Association and Blackwell Publishing Ltd. DOI= 10.1111/j.1467-8659.2009.01405.x

[34] *Customer Showcase*,

`Autodesk Kynapse Middleware`
`http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=11661385&linkID=11654889`

[35] P. Pontevia, (2008) *Pathfinding is Not A Star*,

`Autodesk Kynapse Middleware`
`White Paper,`
`http://images.autodesk.com/adsk/files/3d_perception_pathfinding_is_not_a_star.pdf`

[36] P. Pontevia, (2008) *Open the Eyes of your Non Player Characters*,

`Autodesk Kynapse Middleware`
`White Paper,`
`http://images.autodesk.com/adsk/files/perception_open_the_eyes_of_your_npcs.pdf`

[37] Bullet,

`http://www.bulletphysics.com/wordpress`

[38] Havok,

`http://www.havok.com`

[39] PhysX,

`http://www.nvidia.com/object/physx_new.html`

[40] Luebke, D. Humphreys.(2007) *How GPUs Work*, Computer, vol. 40, no.2, pp.96-100, Feb. 2007

[41] *Nvidia CUDA Programming Guide Version 1.0*, Technical Brief, Nvidia Corporation, June 2007.

[42] Phuong H. Ha, Philippas Tsigas, Otto J. Anshus. (2009) *The Synchronization Power of Coalesced Memory Accesses* , IEEE Transactions on Parallel and Distributed Systems, IEEE Computer Society. (2009-08-13), doi = 10.1109/TPDS.2009.134