Artificial Intelligence in Simulations

ARTIFICIAL INTELLIGENCE IN SIMULATIONS

BY

DAVID HYNES, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2012)   McMaster University

(Computing and Software)   Hamilton, Ontario, Canada

TITLE: Artificial Intelligence in Simulations

AUTHOR: David Hynes

                    B.Eng., (Software Engineering)

                    McMaster University, Hamilton, Ontario, Canada

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: x, 100

Abstract

Simulator systems are becoming increasingly popular within the automotive industry. Driving simulations are used to train new drivers, and to research and improve automobile related technologies. While recent technological advances have made simulators more affordable, they have also made simulators more complex. At McMaster's motion simulator laboratory, we wished to create a system to analyze how drivers behave when behind the wheel in a variety of situations. To accomodate this, we have created a complex, robust system capable of presenting the driver with customized scenarios, and measuring their reactions to those scenarios using a variety of standard psychology techniques, such as EEG. Our system utilizes a variety of software components such as scenario management, artificial intelligence, image generation, and content creation. This thesis describes the use of one software component, the Presagis AI.implant. The AI.implant is an artificial intelligence system used to control autonomous characters within the simulation, which the driver can interact with. The AI.implant has also been extended to allow for more possible experiment scenarios, and to improve the quality of the simulation. These extensions include: the addition of signal lights for vehicles, emergency vehicle characters, and improved character movement at intersections. This thesis demonstrates that an artificial intelligence is a useful component of a more complex simulation, in order to promote further

research in this field.

Contents

List of Figures

Chapter 1

Introduction

The term artificial intelligence (AI) refers to the intelligence of machines, and the fields of scientific and engineering research that strive to develop that intelligence. The design of AI systems focuses on the creation of intelligent entities that are capable of perceiving their environment and reacting to various stimuli. These AI systems vary widely in their functionality, depending on what role they were designed to fill. Some AIs control a single entity, while others control hundreds simultaneously; some AIs interact with hardware, while others are purely virtual. Because of this versatility, AI systems have an ever increasing array of applications, and have been utilizied in the fields of computer science, manufacturing, finance, healthcare, and more.

Since AIs have such widespread use, it is impossible to find a "one size fits all" AI solution. To a certain extent, AI systems need to be customized and adapted in order to fulfil unique requirements. One popular use of artificial intelligences is in video games. These applications produce a virtual world for the player to interact with, and frequently that world will be populated by several characters not controlled by the player (often called "non-player characters" or "NPCs"). These NPCs are

controlled by an AI, which examines the environment and instructs characters how to behave. In this way, AIs produce the illusion of intelligence in these characters. Game AI systems draw on several fields of computer science such as optimization, pathfinding, and animation. They control multiple characters, and the interactions these characters have with their environment, with players, or with each other. They make use of environmental features such as roads, sidewalks, crosswalks, buildings, trees, elevation, and more, to further enhance the behaviours of those characters, enabling them to react to their surroundings.

These game AIs can also be used in simulations. Like games, simulations create a virtual world with which the user can interact. Game AIs are used to populate that world with characters to make the simulation seem more like the real world. However, simulations do differ from video games. Whereas games are used for entertainment, simulations are used for training and research. Because of this, simulations must be realistic in order to be of use, and the AI controlled characters used in simulations must emulate their real world counterparts as closely as possible.

At the McMaster motion simulator laboratory, we sought to create a simulation system in which we could create customized scenarios to examine the reactions of drivers to certain stimuli. This required the simulation to include autonomous characters, which were used to represent vehicles and pedestrians. To meet this requirement, we integrated a commercial "off-the-shelf" AI system (the Presagis AI.implant) into our simulation. In addition to the default functionality of the AI.implant, several additions were made both to provide new possible test scenarios, and to make the simulation more realistic. Firstly, vehicles controlled by the AI.implant had no implementation of signal lights, such as brake lights and turning indicators. Additionally,

the motion of vehicles through left-hand turns at intersections appeared choppy and unrealistic. These issues needed to be addressed to provide the driver with a high quality simulation. Finally, the AI.implant lacked any emergency vehicle characters, such as ambulences or fire trucks. These vehicles provide several possible test scenarios where a driver's response can be measured and studied, and as such, needed to be implemented.

In order for these additions to be fully explained, an understanding of the Presagis AI.implant must be established. First, the basics of game-based AI systems must be discussed in order to better comprehend the conventions and algorithms common to all such AIs, and how they fit into more complex simulation systems. Next, the specifics of the AI.implant need to be examined to discern how this specific system functions. Finally, the explanation of how the AI.implant was extended to include new functionality needs to be included, in order to better understand how our system was improved.

Chapter 2

Background: Game AI and Simulation

In the video game industry, artificial intelligence (AI) systems are used to convey the appearance of intelligence in non-player characters. These systems control the actions of characters which interact with the environment, other characters, or the player. AIs are a critical component of modern games, as they are used to build a sense of immersion in the player, and to facilitate gameplay. The functionality of these AIs can also be applied to simulations, where large numbers of intelligent characters need to be portrayed in a realistic fashion.

The appearance of intelligence in characters is the result of several elements of the AI. The actions characters perform are represented by behaviours. Each behaviour added to a character gives it a new set of tools with which to interact with the world around it. Specifying a character's behaviour is not enough, however. Characters must also be able to navigate their world intelligently, otherwise they would not be able to move through their world realistically.

These and other elements must be brought together in order for an AI system to accurately describe the properties and actions of the real world entities that the intelligent characters are meant to emulate. Fortunately, AI systems have a number of tools and algorithms available to them to help achieve this feat.

2.1   Characters

Characters are the various objects used to simulate conscious interactions with the environment, or with other characters. They mimic the actions of humans, animals, vehicles, or similar creatures or constructs. Characters occupy a location in the simulation and can move around the world according to certain rules of navigation which can be set individually for each character, allowing different characters to represent different types of real world entities. Characters will also react to different stimuli in different ways, depending on their specific implementation.

Characters are attributed with certain properties. These properties are used both to describe the bounds and functionality of a character in the world, and to differentiate characters from each other. The properties of a character affect how it behaves in the world. Simple properties include values such as character size, speed, or position. More complex properties may include values such as turning rates and maximum speed, or vector values such as orientation. Character properties are customizable and can be fine tuned depending on how the character is meant to act in the world, or what kind of entity the character is meant to emulate.

2.2   Game AIs in Simulations

While game AIs are useful in simulations, the environment of a simulation differs from that of a game. On the surface, simulations and games do appear similar. They both present the user with a three-dimensional environment, which they can interact with by manipulating a user interface. However, there is a fundamental difference between games and simulations. Games are used primarily for entertainment. As such, the act of playing a game is often nothing at all like performing the actual task the game describes. Consider as an example, a simple car racing computer game. At a high level, the game seems similar to driving an actual race car. The player is in a car and controls it as it moves around a track. However, operating the game is not the same as operating an actual race car. The player controls the game through a mouse and keyboard, or controller, rather than physically steering the vehicle and feeling the motion of the car. The game presents an approximation of driving used only to entertain, not to train new race car drivers, or research the effects of driving on humans.

Simulations on the other hand, strive to give the user an experience similar to what they would expect in the real world. Using the previous example, if we sought to train a new race car driver by using a simulation, that simulation would have to be similar to the real world vehicle in order for anything the driver learned in the simulation to be applicable to the real world. This presents new challenges to game AIs, as the autonomous characters they control must behave reliably and realistically in order for the simulation to be an accurate portrayal of the real world.

In the McMaster motion simulator, subjects are examined to determine how they react to various stimuli while driving. Again, in order for this data to be usable in

the real world, the driving simulation must be similar to a real world automobile. Our system utilizes the Presagis AI.implant to control the autonomous pedestrians and vehicles the driver interacts with on the road. The presence of these characters improves our system, both by making the simulation conditions similar to everyday driving conditions, and by providing new scenarios in which to examine drivers' reactions to stimuli. This makes the AI.implant a critical component of our driving simulation. To fully understand how the AI functions, it is necessary to first discuss how it fits in as a part of a larger, more complex simulation.

2.2.1 Structure of the Simulation

Our system follows the typical layout of a commercial simulator. The core of the system is comprised of the platform simulator, the image generator (IG), the artificial intelligence, and the scenario manager. Each of these components is responsible for a different task.

1. **Platform Simulator**: the physics system that the human driver utilizes to control their own vehicle.

2. **Image Generator**: responsible for rendering the simulation as a 3D image on the screen.

3. **Artificial Intelligence**: used to control autonomous intelligent characters.

4. **Scenario Manager**: acts as a controller by managing the communication between other components.

A number of protocols are used to facilitate communication between the different system components. The scenario manager communicates with the platform simulator

and the artificial intelligence using two custom built protocols. The scenario manager communicates with the IG using the Common Image Generator Interface (CIGI). CIGI is a standard interface used to facilitate communication between an IG, and its host simulation.

Since the IG must update the screen several times each second, it is what drives the timing cycle for the rest of the system. When the IG determines that a new frame needs to be rendered, it notifies the scenario manager. The scenario manager then communicates with the platform simulator and artificial intelligence to determine the new positions of all the characters in the simulation. The scenario manager sends this information to the IG, which renders the characters at their new positions.
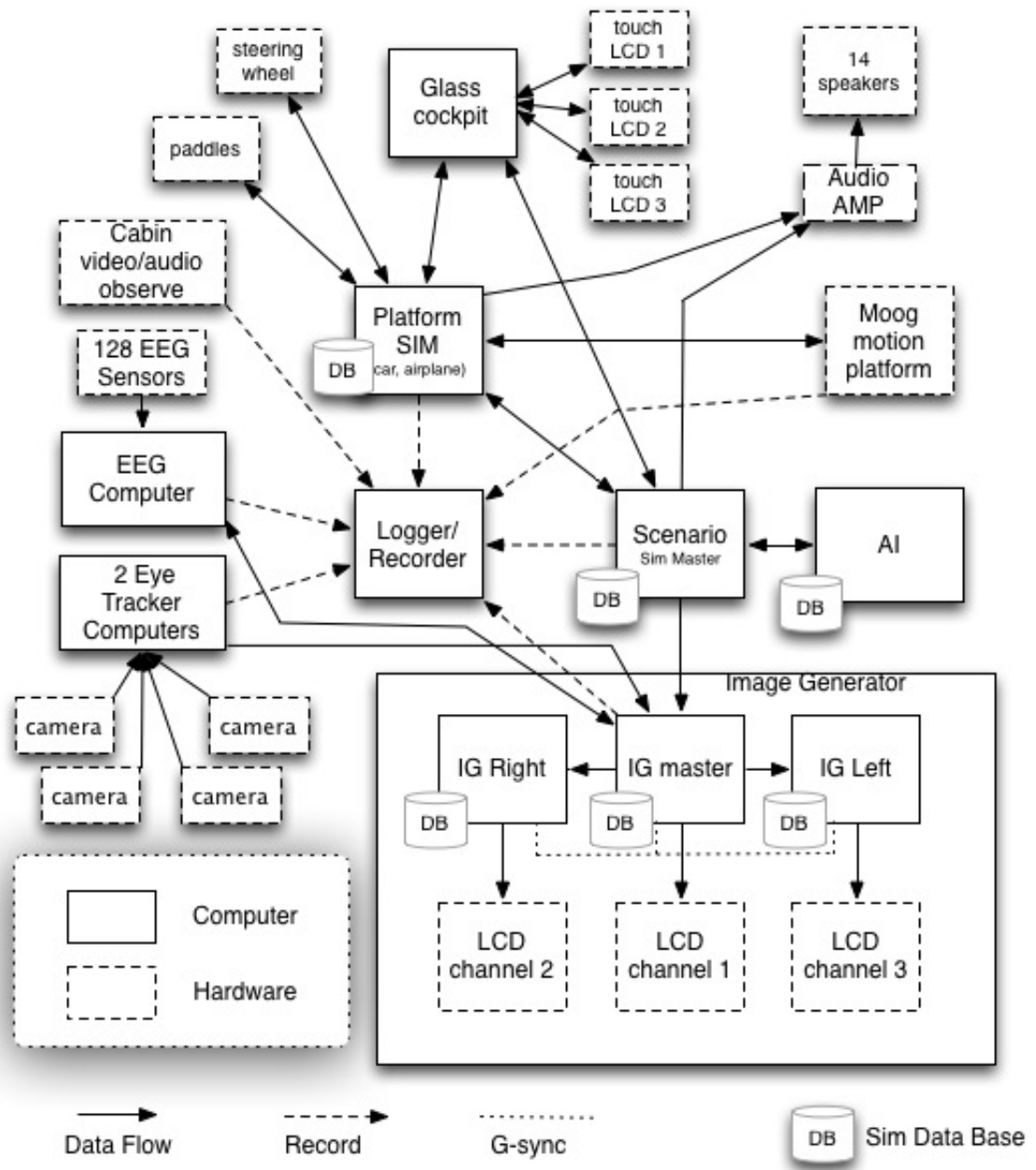
Figure 2.1: The AI as a component of a larger simulation system.

2.2.2   Integration of the AI

The AI is connected to the rest of the simulation through the scenario manager. Since the scenario manager must update the IG with the positions of characters, it must know where each character is in the world, including those controlled by the AI.

Since the AI's autonomous characters and the human controlled vehicle are capable of interacting with each other, the AI needs to know the position of the driver, and the platform simulator needs to know the positions of any autonomous characters. The AI will update the scenario manager with the new positions of all the autonomous characters it controls, and the scenario manager updates the platform simulator with this information so it can react correctly to the presence of these characters (in a collision, for example). Similarly, the platform simulator sends the position of the driver to the scenario manager, which in turn updates the AI. This allows the AI's autonomous characters to respond to the driver in accordance with any behaviours they possess.

When the scenario manager has been updated with the positions of all characters, it then updates the IG which renders these characters in the 3D environment on the screen.
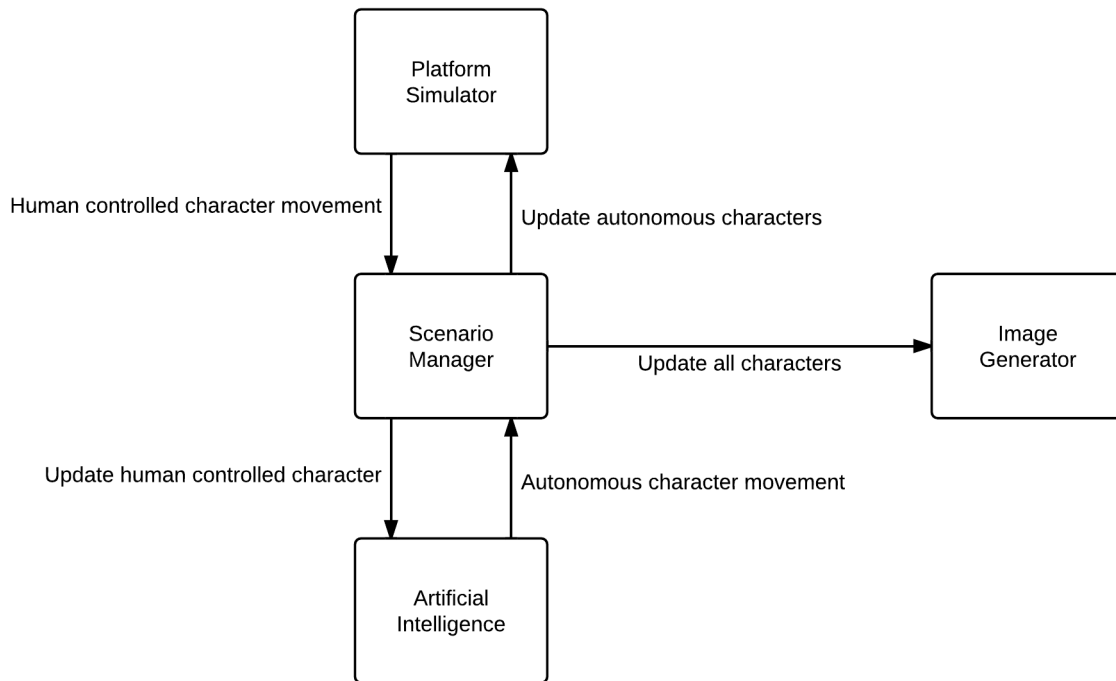
Figure 2.2: The flow of information to and from the artificial intelligence.

2.3   Motion

In (Reynolds, 1999), motion is divided into a 3 tiered hierarchy. These layers have been labelled with different titles depending on the setting, but Reynolds refers to these layers as: action selection, steering, and locomotion.

Locomotion is the lowest layer in the hierarchy, and corresponds directly to the character's movement. The locomotion layer describes how an individual character moves through various properties such as mass, speed, and turning rates. Because of this, locomotion is dependent on the character's constraints. Since two different types of characters will have different properties and constraints, they will therefore have separate locomotion schemes.

The **steering** layer represents the methods the character uses to reach its goal. A steering force is applied to the character which is used to direct it towards its goal. In the steering layer, the path towards the character's target is calculated, and the force that moves the character along that path is applied. Because steering is achieved by applying this force rather than adjusting the character's properties, the steering layer is independent of the character. Steering methods can be easily transplanted onto different locomotion schemes and still peform similarly. The steering layer is also where most of the character's navigation takes place.

Action selection is the highest layer in the motion hierarchy. It represents the higher level behaviour logic which determines the motion strategy and the character's goal. This layer determines the character's target and sets the character's goal to either move towards or away from its target. The action selection depends primarily on the behaviour the character is emulating.

2.4   Navigation

Character movement is critical in order for characters to achieve their desired goals. However in simulation environments, which have the intent of conveying a strong sense of realism, motion by itself is not enough. Intelligent navigation is required for characters to move fluidly and naturally throughout their world. A character's ability to navigate correctly will enable it to move through much more complex environments.

Characters use a variety of navigation methods to achieve their desired goals. For example a character can navigate around walls and obstacles in order to seek its target. In another example, a vehicle character can navigate through an urban environment by moving down streets while avoiding other vehicles and pedestrians.

These examples however, are still fairly simple. There are several navigational tools which can be used individually or in conjunction with each other to achieve realistic navigation in a much more complex environment.

2.4.1   Pathfinding

Pathfinding refers to the way in which a moving character will calculate a path through the world towards its target. Smooth pathfinding is a fundamental problem faced routinely in the production of modern commercial video games (Goodwin *et al.*, 2011). Games typically require a large number of characters with the ability to navigate correctly around varying environments. Because of this, pathfinding in video games has been researched for many years (Cui and Shi, 2011). A common method of pathfinding is to divide the world into some form of grid or network. Cells of the grid or points of the network can then be viewed as nodes of a search graph. While the graph representation is not an accurate model of the world (the graph does not convey spatial information such as size or proximity), the graph does convey the connectivity of the grid nodes and can be searched using a variety of algorithms to compute the optimal path (Goodwin *et al.*, 2011).
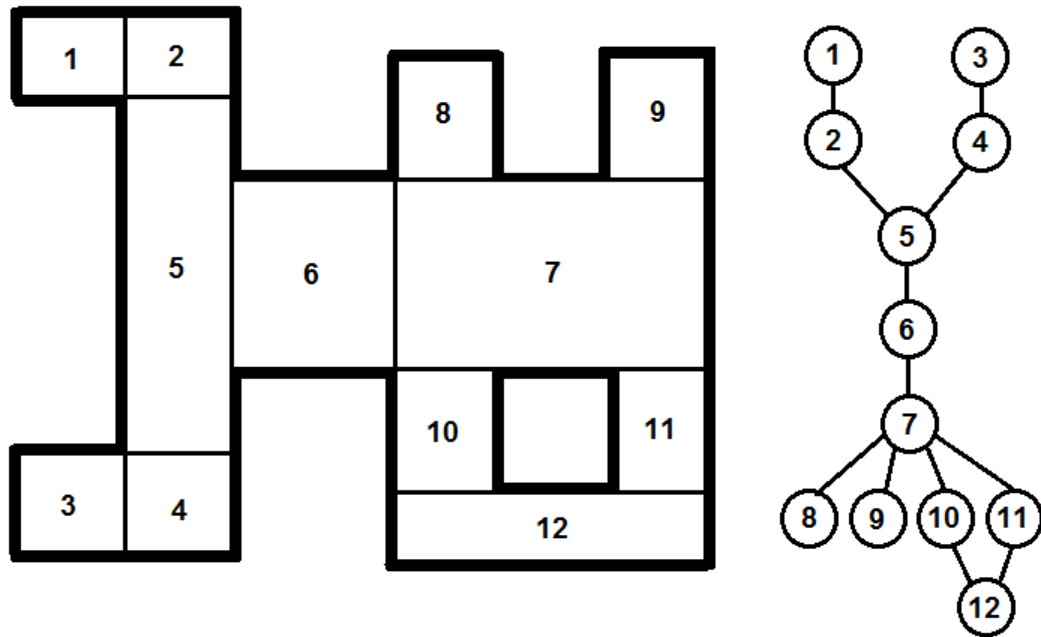
Figure 2.3: A sample world divided into a grid. The grid can then be represented as a connectivity graph which can be searched to determine the optimal path.

Initial solutions to pathfinding included several algorithms for searching the connectivity graph, such as the depth first search, breadth first search, and Dijkstra's algorithm. However, as technology progressed, these algorithms were quickly overwhelmed by the exponential growth in complexity of games (Cui and Shi, 2011).

In modern games, one of the most popular pathfinding algorithms is referred to as the A* (A-star) Search Algorithm (Goodwin *et al.*, 2011). The A* algorithm works by assigning each node in the graph two integer values, the first representing the cost to move to that node from the start node, and the second representing an estimate of the cost to move from that node to the target node (this estimation can be calculated using several methods). The algorithm begins at the start node and examines all adjacent nodes to find the node with the lowest sum of these two values, then begins examining that node's adjacent nodes. The algorithm may recalculate the values of

a node, if it encounters that node more than once through different routes, to see which route through that node is most optimal. Eventually the algorithm ends when either all nodes have been examined (in which case no path is possible) or when the target node is found (Lester, 2005).

The A* algorithm is a generic search algorithm which has optimizations available to improve efficiency. The algorithm itself however has several advantages alone. A* is guaranteed to find a path to the target if such a path exists, and it is optimal if the estimations for the remaining distance to the target are admissible (meaning the estimates are equal to, or less than, the actual cost) (Cui and Shi, 2011).

The use of this and other algorithms, however, depends on how the world is represented internally. The world must be broken down into nodes of some description in order for a graph representation to be viable. Fortunately, many solutions to this problem exist and are used in a variety of applications.

2.4.2 Waypoint Network

A waypoint network is a simple tool for aiding navigation. A waypoint is an object that occupies a position in the world which characters can detect. Waypoints do not have a visual component in the final simulation, and are typically not considered collidable, meaning characters can pass through them. They are used solely for the purpose of aiding navigation. A waypoint network consists of multiple waypoints linked together by edges. The edges represent the path between waypoints that characters will follow when they use the waypoint network. Characters will move between waypoints as they attempt to seek their targets. However, their target may not always lie on the waypoint network. If this is the case, the character will move to

the waypoint in the network that is closest to its target, thereby maximizing the time it spends on the network. Once this waypoint has been reached, the character will leave the network and move directly towards their target until it is reached (assuming the target is not obstructed in any way). If the character then picks a new target, it will move back to the waypoint network to resume navigation.

While waypoint networks work well for simple areas, they do suffer drawbacks in other situations. For example, waypoint networks become unwieldy in large open areas. In order to produce realistic movement over a large area using a waypoint network, a large quantity of waypoints is required. As a result, the resulting path may produce irregular movement patterns, such as zigzagging between waypoints (Tozour, 2008).

Other issues include a lack of path correction around obstacles. If a waypoint network becomes obstructed, characters navigating on it may become stuck. Characters cannot move away from the network to avoid the obstacle because they cannot analyze the geometry of the world. This could result in characters falling off the world geometry or becoming trapped (Tozour, 2008). Furthermore, waypoint networks can fail for different types of characters (Tozour, 2008). For example, a waypoint network may pass through a small door which a small character can pass through, but a larger character cannot. If the larger character is not given additional pathfinding information about this area, the character's navigation may fail.

To help alleviate some of these issues, a more robust form of navigation called a navigation mesh, or navmesh, can be used instead of, or in conjunction with, a waypoint network.

2.4.3 Navigation Mesh

A navigation mesh, or navmesh, is a collection of polygons that are used to represent the ground that various characters can navigate on. Navmeshes partition a surface into a collection of convex cells. These cells are usually triangular in shape, but not necessarily so. Characters can move between two points in a cell, or across cell edges to enter adjacent cells. Characters cannot, however, move across an edge and leave the navmesh, then return to the navmesh in another cell. In this way, character movement is limited by the bounds of the navmesh.
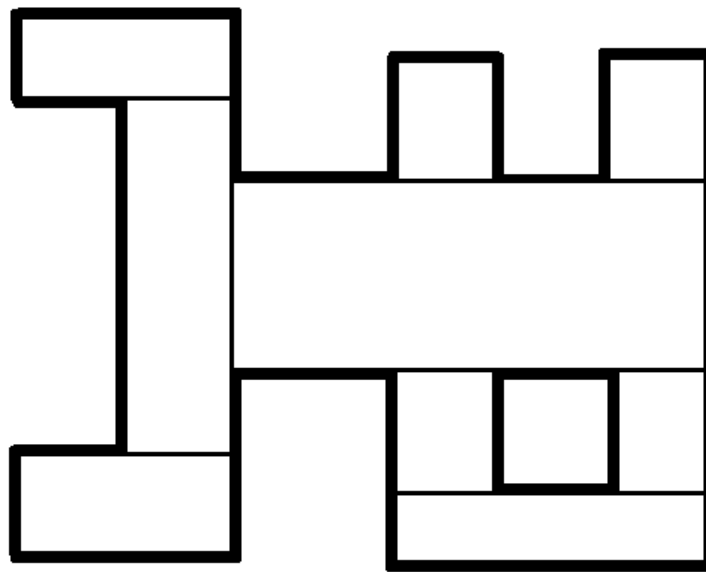
Figure 2.4: A simple navmesh composed of a geometry divided into polygonal cells.

Pathfinding on a navmesh can be seen as similar to pathfinding on a waypoint network, but the waypoint objects have been replaced by polygonal cells (Tozour, 2008). Depending on the size of the polygons, some waypoint nodes can be removed entirely.
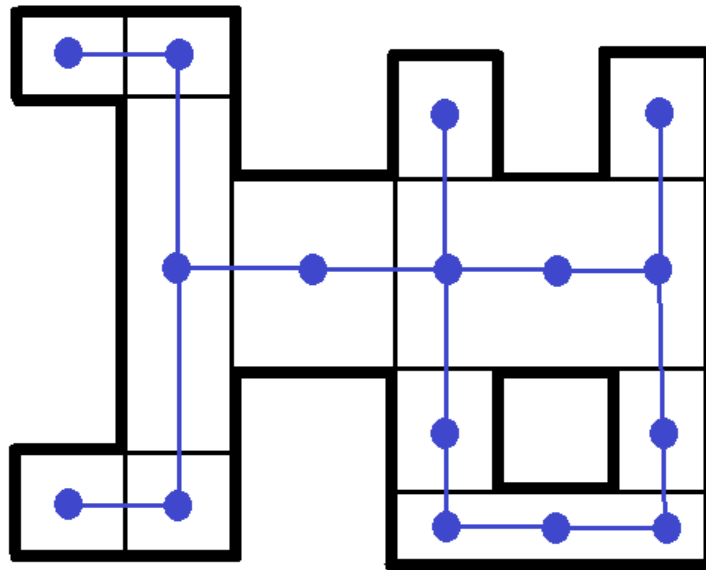
Figure 2.5: A navmesh is similar to a waypoint network, but with polygons at the nodes. Some nodes may become unnecessary.

Because the cells of a navmesh are convex, a character can move between two points in the same cell without the risk of crossing an edge. If a cell is not convex, then there can exist two points on that cell that the character cannot move between without either crossing an edge they normally would not be allowed to cross, or receiving extra pathfinding information which cannot be provided by a single cell.

(a) A Convex Cell   (b) A Non-Convex Cell
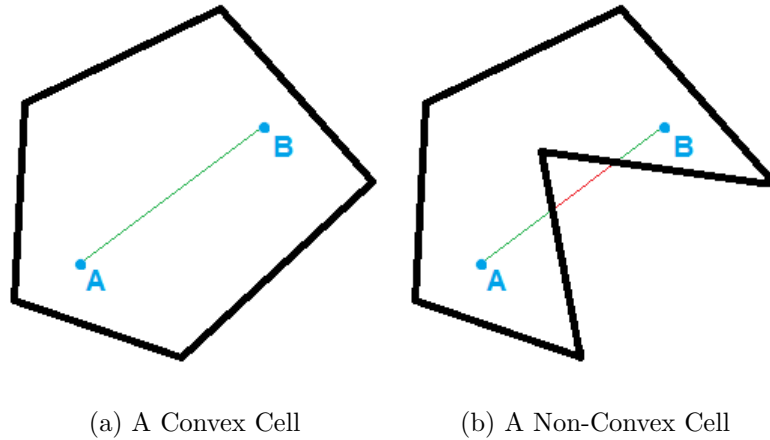
Figure 2.6: Any two points in a concave cell can be connected without crossing an edge. In a non-convex cell, there exists two points that cannot be connected without crossing an edge, or without additional pathfinding information.

To navigate, a character begins on a cell with its target lying somewhere on the navmesh, also in a cell. If the character is on the same cell as the target, it can simply move directly to it because the cells are convex. However, if the character and target are on different cells, a path across the navmesh must be constructed. The character determines the next cell it must move to, by searching the connectivity graph, and enters that cell. This is repeated until the character and target lie in the same cell.

Navmeshes are not flawless however. The main problem of navmeshes comes in the form of a trade off between the quality of the path that is found, and the size of the search space (Goodwin *et al.*, 2011). For example, if a navmesh uses cells that have a large surface area, the total number of cells in the navmesh is reduced, thereby reducing the size of the corresponding graph, and reducing the amount of cells that must be searched in order to calculate the path. However, when navmesh cells become too large, the quality of the path calculated is affected.
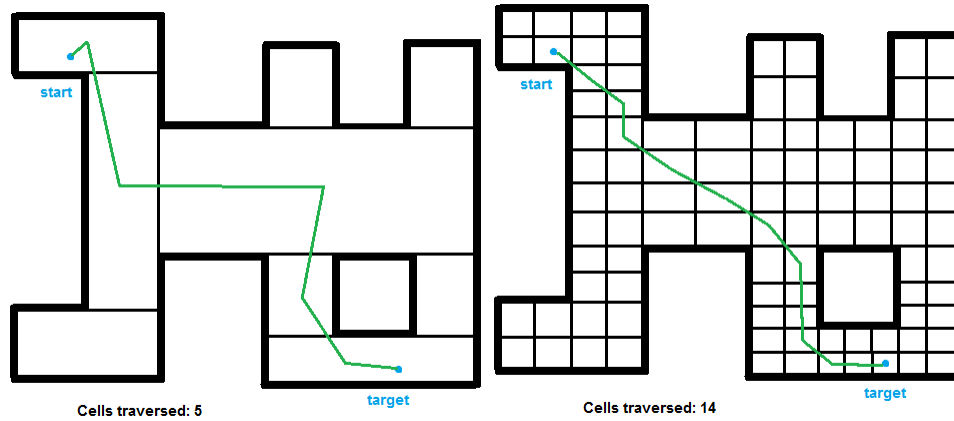
Figure 2.7: Navmeshes with large cells have fewer cells to traverse, but poorer path quality. Conversely, navmeshes with small cells have higher quality paths, but pathfinding algorithms must search more graph nodes.

To combat this, a balance must be struck between path quality and algorithm efficiency. However, where this balance is struck depends on the specific environment the navmesh is being built on.

2.4.4 Quadtrees

One method to reach this balance is the use of quadtrees. At a high level, quadtrees are very similar to navmeshes. They divide the world up into navigable and non-navigable convex cells which characters can move across. Where quadtrees differ from standard navmeshes is in the creation of those cells.

A quadtree divides the world into four equally sized squares. Each of these squares that contains an obstacle or barrier is then subdivided again into four smaller squares. This process is continued recursively until either none of the navigable squares contain obstacles, or the area of the squares has reached a predetermined minimum.

Figure 2.8: A section of a world that has been divided into cells using a quadtree approach.

The advantage of using the quadtree approach is that large unobstructed areas can be represented with fewer cells. This reduces the search space of pathfinding algorithms. The disadvantage is that in these large unobstructed areas, the quality of the path is reduced (Goodwin *et al.*, 2011). An additional disadvantage is that if the minimum cell area is too large, some obstacles may not be accurately represented during pathfinding.

2.4.5 Blind Data

Blind data is a set values stored within the polygonal cells of a navmesh, which are used give those cells certain characteristics. Blind data is used to assist in character pathfinding, as characters on the navmesh can examine the blind data and thereby determine certain properties of a cell. For example, blind data is used to determine which areas are navigable and which are non-navigable. Multiple blind data types can be stored in the same cells. This allows for more complex navigation scenarios, where characters with different pathfinding requirements are able to navigate on the

same navmesh.

Figure 2.9: A navmesh attributed with blind data. Blind data can be used to differentiate between navigable and non-navigable cells on a navmesh.

The primary use of blind data in this setting is to specify the differences in navigable surfaces (Kruszewski, 2006). Doing so allows characters to navigate correctly. A common example of this is an urban street setting with features such as roads, sidewalks, and crosswalks. These surface types are represented by the same navmesh, but different characters navigate on that navmesh differently. A car for example, should navigate on roads but not on sidewalks, while a pedestrian must only cross roads at crosswalks. Blind data can be utilized to accomplish this functionality. Blind data can be assigned to each of the cells in the navmesh and used to differentiate between cells which represent roads, sidewalks, and crosswalks. The pathfinding of individual character types can then be constrained to ensure pathfinding occurs as intended. A pedestrian character can be constrained to only navigate on cells which contain the blind data value associated with sidewalks and crosswalks. Conversely, a car character

can be instructed to only navigate on cells which are marked as roads and crosswalks.
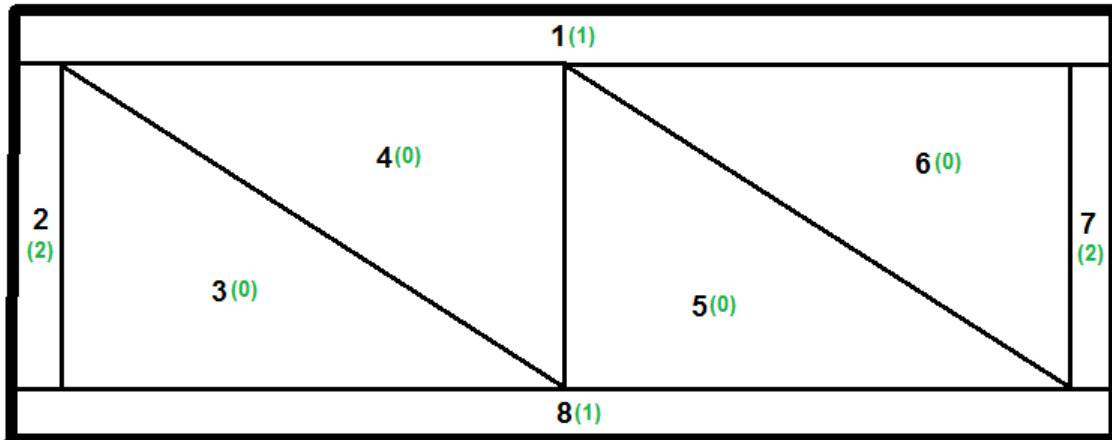
Figure 2.10: A sample use of blind data. Different values differentiate between sidewalks, crosswalks, and roads.

2.5   Behaviours

Behaviours are the general set of rules that describe how a character performs. Each behaviour is meant to simulate a specific action or set of actions that may be undertaken by various entities in the real world. Behaviours typically have several parameters that can be fine tuned to produce the optimal desired outcome. For example, a behaviour that instructs one character to move to a target position may have a parameter which dictates the speed at which the character should move. Behaviours are also limited by the character performing them. For example, if a character has a parameter describing the maximum speed at which it can travel, then any behaviour that character performs cannot cause the character to exceed that maximum speed. Behaviours affect the motion of the character by calculating a steering force. The steering force's intensity and direction depend on the specific calculations performed

by the behaviour. The steering force is applied to the character to cause it to move towards its goal (Reynolds, 1999).

2.5.1   Combining Behaviours

Each character can have multiple behaviours assigned to it thereby combining their functionality. This allows simpler behaviours to serve as components of more complex interactions. A simple example is to assign a target seeking behaviour, and an obstacle avoidance behaviour to a character to allow that character to seek its target while avoiding other characters and objects.

There are two ways in which behaviours can be combined which are outlined in (Reynolds, 1999). The first method is to sequentially switch between active behaviours depending on the character's current circumstance. For example, a character may seek a target normally, but switch to an evasive behaviour if another character comes within a certain radius. This method is useful in siutations where a discrete behavioural switch occurs, but is not effective in situations where multiple behaviours need to be utilized simultaneously, or where the trigger event which signals the switch between behaviours is not easily identifiable.

The second method of combining behaviours is to blend them together. This blending can be done in several ways. The simplest blending method is to compute the steering force vectors from each of the behaviours and sum them together. By using a weighting factor for each of the steering forces, some behaviours can have a stronger influence over others. While simple, this method does have several drawbacks. A common drawback is that even with adjusted weights, certain behaviours may simply cancel each other out.

Another method of blending can resolve this issue. Instead of summing all the component behaviours together, each behaviour can be assigned a unique priority. In this case, the highest priority behaviour is the one which is used normally. However, if this behaviour returns a zero vector as the steering force, the next highest priority behaviour is used, and so on.

An additional method of blending outlined by Reynolds (Reynolds, 1999) is referred to as "prioritized dithering". It functions similarly to a priority based blending method, but each behaviour has a pre-defined probability of it being used. If the highest priority behaviour returns a zero result, or is skipped over by the random selection, the second priority behaviour is evaluated, and so on.

Chapter 3

The AI.implant

The McMaster motion simulator utilizes the Presagis AI.implant as its artificial intelligence system. The AI.implant is a commercial artificial intelligence for the simulation of individual humans, scalable to hundreds of people in real-time. This is achieved by utilizing several techniques and concepts used in game-based intelligences, as the video game and entertainment industries have become a driving force behind simulated humans and graphics (Kruszewski, 2006). The AI.implant has a particular focus on simulating realistic urban environments and the behaviours of human agents (such as pedestrians or motorists) within those environments. This makes it ideal for use in a simulation of an urban setting where a user interacts with potentially hundreds of digital characters, each with their own intelligence.

3.1 Attributes

Attributes are extremely important components of any object in the AI.implant. Attributes are the properties of an object, such as radius or speed, that can be

adjusted to fine tune the appearance or functionality of that object. Attributes are present on all characters, navigation tools, behaviours, and various other objects. When a new object is created as a more specialized version of a pre-existing object, the new object inherits the attributes of its predecessor and may add its own unique attributes. This helps to ensure that any more specialized version of an entity contains the original functionality of that entity.

Attributes can be stored as a variety of data types, and can represent a variety of values. For example, an object's radius can be stored as a floating point attribute, or an object's name can be stored as a string attribute. Attributes can also be boolean values used to toggle certain functionality or properties, such as whether an object is considered collidable.

Attributes are initialized before runtime, but they can also be modified during execution. This allows various objects to be tuned dynamically, so they may adapt to more complex environments.

3.2 Characters

In the AI.implant, objects in the simulated space fall under one of two categories: characters, or the world. Characters are the objects that perform actions, such as moving, while the world is simply the environment that surrounds the characters and the objects that make up that environment (Kruszewski, 2006).

Characters are intelligent entities which populate the world. They can be divided into the categories of autonomous and non-autonomous. Autonomous characters are the characters which are under the AI.Implant's control and act independently, according to the rules which they are given. They can be controlled on a character

to character basis, where each character is given its own goal and means with which to reach that goal, or they can be controlled at a higher level as a group.

Autonomous characters have three key ways in which they demonstrate intelligence (Kruszewski, 2006):

1. Moving through the world while avoiding collisions with static objects or other characters.

2. Reacting to environmental stimuli by making decisions.

3. Animating (moving limbs and body parts) in a convincing manner.

Conversely, non-autonomous characters are characters which are controlled by various other external controllers, such as human operators (Kruszewski, 2006). In the McMaster motion simulator, the human driver is represented in the world as a non-autonomous character.

3.2.1   The Basic Character

The basic character is the most general representation of entities within the simulation. It contains the general attributes that all characters of any type must include. Each character has its own pathfinding service, and obstacle avoidance service and as such, contains its own path manager and avoidance calculator. These tools are used to aid the character's navigation through its environment by keeping the character away from objects identified as obstacles, and keeping the character in areas that it is allowed to move through. Having these services attached to the basic character ensures that they will be available to any of the more specialized character types.

The AI.Implant's characters are similar to the simple vehicle model described in (Reynolds, 1999). All characters are based on a point mass approximation. Since a point mass itself is not an accurate description of any real world entities, character size is increased with a specified radius which is stored as an attribute. Other important attributes related to characters include the character's current speed and direction, which change depending on the character's actions, as well as attributes such as maximum speed and maximum acceleration, which are constants used to limit the mobility of the character.
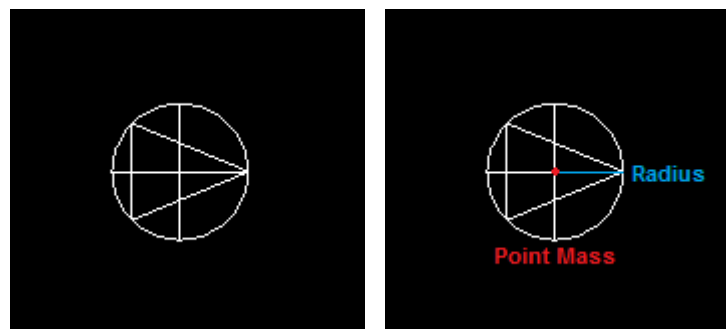
(a) Basic Character   (b) Character Dimensions

Figure 3.1: The generic character.

3.2.2   Important Character Attributes

The basic character is the foundation upon which all other characters are based. As such, the attributes of the basic character are also shared by the other character types. Many of these shared attributes are simple to understand, such as speed or direction. Other attributes are less obvious in their effects, but are nonetheless crucial in ensuring the characters behave as intended.

ID

Every object in the simulation that the AI manages, including characters, behaviours, and even static obstacles, shares an attribute called the ID. The ID is an integer value which enables the AI.implant to easily identify every object in its internal inventory. Each object's ID is unique, so that the ID can be used to differentiate between objects, even if some objects are exact duplicates of one another.

Navmesh ID

The navmesh ID attribute holds the ID of the navigational tool (such as a navmesh or waypoint network) that the character is using to navigate the world. The character uses this to determine which navmesh or network it is meant to respect. The ability of the character to select the proper object is crucial in environments with multiple navigable areas.

Frustration

One of the character's most important attributes, with respect to behaviour, is frustration. Frustration is a floating point value which represents the character's frustration at being unable to progress along its desired path due to obstacles impeding it. A default frustration value (usually 0) is set before runtime. This value increases as the character is forced off of, or blocked from, its desired path, and decreases as the character progresses normally.

Frustration can be used to impact the actions of a character. A high level of frustration is indicative of a character whose desired behaviour is being impeded in some way. The character can determine when its frustration has become too high,

and find a new path, or take some other action which will enable it to bypass the obstruction. This allows characters to perform their required actions more reliably. Frustration can be especially useful in traffic simulations to simulate the tendancies of certain vehicle types, and to maintain a more reliable traffic flow.

Clearance

Clearance refers to the amount of space the character keeps between itself and any walls or barriers while navigating. While the character normally attempts to maintain this clearance, this attribute may not be strictly adhered to. A character may move closer to a wall or barrier if it is required to do so in order to achieve its assigned goal or to avoid a collision (Presagis, 2011b). This attribute may also be used in conjunction with another attribute called "respect clearance", which if toggled will cause the character's pathfinding routines to avoid areas that may be too narrow to accomodate the character when using its defined clearance (Presagis, 2011b).

3.2.3 Character Types

Autonomous characters are again subdivided into two categories: vehicles and persons. Persons are very similar to the more general character. The person adds only a few attributes which aid in navigation in an urban setting. These attributes are used to enable the person to identify crosswalks, and respect traffic rules, such as stopping at red lights.

The vehicle character type is much more specialized. Vehicles include all the attributes associated with a character, and introduce new attributes as well. These attributes include:

- **Driver type**: refers to the preferred lane the vehicle will use on multi-lane roads (fast, medium, or slow).

- **Respect speed limit**: how the driver will respond to speed limits. It can be set to strictly respect limits, respect limits within a certain range, or ignore limits.

- **Max speed ratio**: the maximum factor by which the vehicle will exceed the speed limit if the respect speed limit attribute is set to respect within range.

- **Min speed satio**: The minimum factor by which the vehicle will move below the speed limit if the respect speed limit attribute is set to respect within range.

- **Pass frustration threshold**: The level of frustration at which the vehicle will attempt to pass the vehicle in front of it using a slower lane.
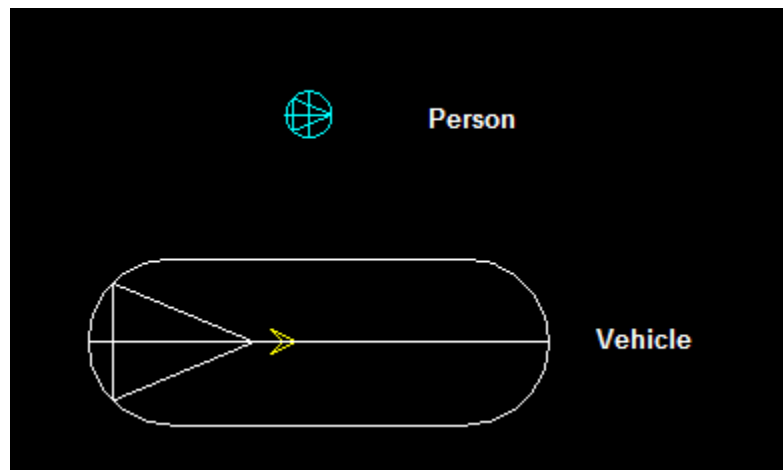

Figure 3.2: A visual comparison between person and vehicle characters. Persons are similar to the generic character, while the vehicle takes a larger, more distinct shape.

In addition to the vehicle character, there is an even further specialized character which acts as a simple automobile. It is referred to as the "wheeled" character.

Wheeled characters add a new attribute called the "wheeled type". Wheeled type refers to the type of automobile the object is meant to simulate. It is an enumerated type which can have a value of "mini", "sedan", "SUV", or "other". Selecting one of these as the wheeled type will change the value of other character attributes, such as length, radius, maximum acceleration, and more. For the former three values of the wheeled type, these attributes are automatically set to pre-determined values which cause the wheeled character to take on the characteristics of the real world automobile corresponding to the selected wheeled type. If the "other" value is selected, these attributes can be configured manually.
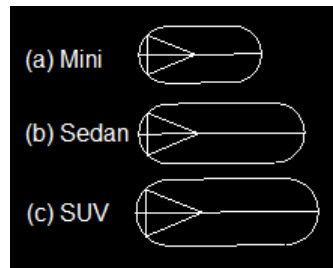
Figure 3.3: Each of the default wheeled types has a unique shape as well as other specific attribute values.

3.2.4 Brain Sharing

To better support worlds with large quantities of characters, the AI.implant makes use of a concept referred to as "brain sharing". The basis of brain sharing is to have multiple characters which share the same intelligence. These characters will behave in the same manner when in the same environment. That is to say, characters sharing a brain will not behave exactly the same as mirror images of each other, but rather that these characters behave the same way in similar environments, and make similar decisions when presented with similar stimuli.

In the AI.implant, brain sharing is implemented via the use of character templates. When a character is created, another already existing character can be selected as the template. When this is done, the new character automatically copies the attributes and decision logic from its template, causing the new character to behave similarly to the pre-existing character. Furthermore, any modifications made to the template character will automatically be propagated to the new character (Kruszewski, 2006).

This functionality allows the AI.implant to scale character simulations into the hundreds. This is because the brain sharing reduces the time required to author simulations with large amounts of characters. Very few original characters need to be created since most characters will simply be clones of a template, and any changes made to the template will cause the clones to be automatically updated (Kruszewski, 2006).

3.3   Navigation Contexts

A navigation context, or navcontext, is the basic interface object used in the implementation of the various tools that characters use to navigate their environment. In the AI.Implant, each navcontext is made up of numbered cells connected together with edges. The AI.Implant uses the basic navcontext to implement navmeshes, waypoint networks, metaconnection networks, as well as road networks, which are more specialized and robust versions of waypoint networks, used to simulate roads on which traffic can move.

3.3.1   AI.Implant Navigation Meshes

The AI.implant uses a simple navmesh implementation. The navmesh is a collection of two dimensional cells connected together by edges, that is used to represent the ground which characters will navigate on. Each cell is given a unique integer that acts as an index. Characters can read the index of the cell they are on, which aids the various pathfinders in computing a path to the character's target.

Navmesh cells can also be assigned blind data, providing additional pathfinding information with which a character's navigation can be constrained. Each cell can have multiple blind data types associated with it, so different types of characters can navigate in different ways on the same navmesh.
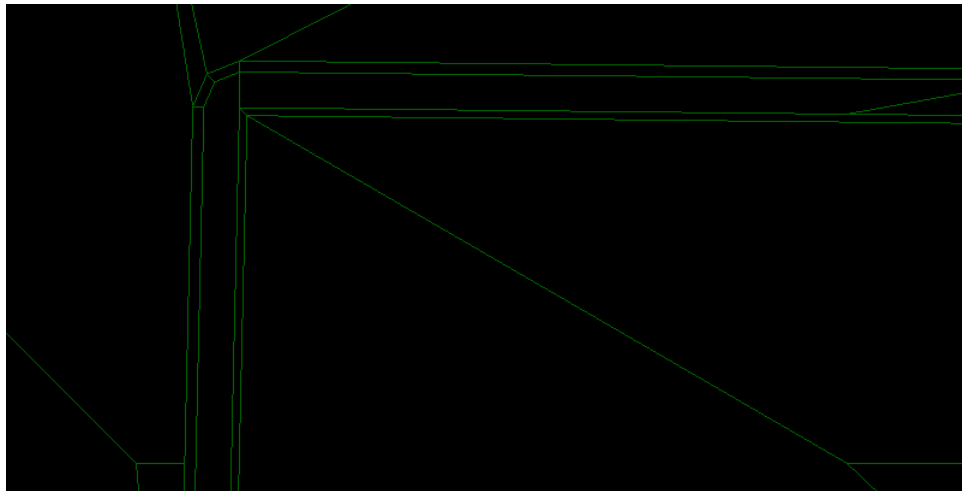

Figure 3.4: A section of a sample navmesh showing multiple cells of varying size and shape. Characters will navigate between adjacent cells.

3.3.2   Blind Data

In the AI.implant, blind data can be added to navigation contexts to improve pathfinding for various characters. Each blind data type is given a unique name, then integer

values associated with that name are placed on cells. This allows one navcontext to support multiple blind data types, allowing multiple character types to navigate on them in different ways.

Blind data aids navigation when used in conjunction with pathfinding constraints. Pathfinding constraints are added to characters to restrict movement across certain areas. Pathfinding constraints describe a range of blind data values, specified by a minimum and a maximum value. A character can only navigate through cells whose blind data value lies within that range. Characters can have differently configured pathfinding constraints for different types of blind data, which allows characters to respect multiple blind data types on multiple navcontexts.

Additionally, to aid in navigating through urban environments, the AI.implant makes use of a special blind data type called the crosswalk tag. Cells on the navmesh with a specified blind data value can have a string ("crosswalk" by default) "tagged" to them. This allows certain blind data values to be referenced by this string rather than by the integer value. The Person character type contains an attribute referred to as the crosswalk tag, which is also a string value. If the person approaches a cell which is tagged with the same string as the person's crosswalk tag, the person will identify that cell as a crosswalk. This causes the person to respect traffic rules, such as traffic lights, while on these cells. For example, when a person finds a cell containing the crosswalk tag, and the traffic light associated with that crosswalk is red, the person will stop and wait until the light has changed before proceeding across the intersection.

3.3.3   Metaconnection Network

When a single navmesh would be too large and cumbersome, a metaconnection network is used instead. A metaconnection network is made up of several navmeshes that are linked together via metaconnections. A metaconnection is a link between two waypoints, with each waypoint lying on a different navmesh. Those navmeshes are then stitched together at the cells along their borders, connecting the cells together. This allows characters navigating on a metaconnection network to move between those stiched cells, crossing from one navmesh to another. The metaconnection network can therefore operate as a large navmesh.

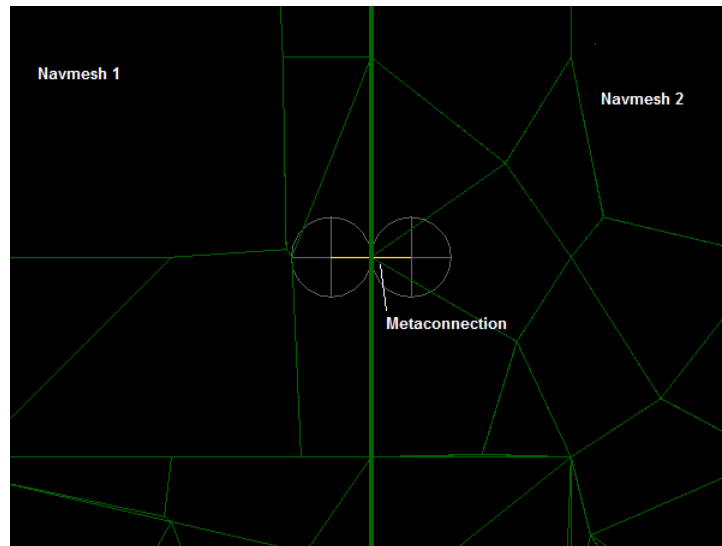Figure 3.5: A metaconnection linking two navmeshes. Cells along the boundaries which have been connected by the metaconnection become stiched together. Characters will be able to navigate between stiched cells.

3.3.4   AI.Implant Waypoint Networks

AI.implant waypoint networks consist of multiple waypoints linked together by edges. A waypoint is centered at a specific point and has a radius that defines the space the

waypoint occupies. Characters move along edges to navigate between waypoints as they attempt to reach their targets. However, characters are not rigidly glued to edges. If the edge is obstructed by some obstacle, characters will attempt to move around it by moving away from the edge they are following.
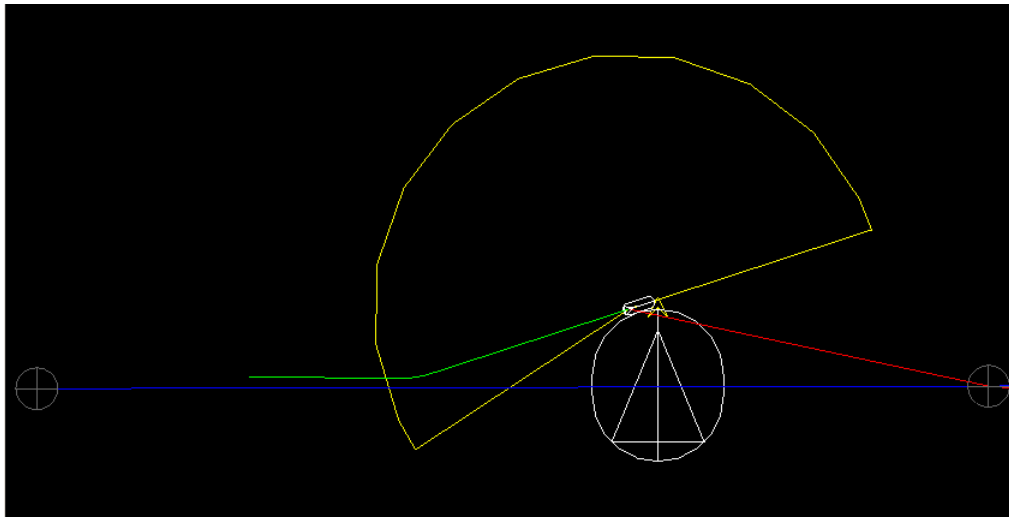
Figure 3.6: Characters will temporarily move away from the waypoint network edge they are following in order to avoid obstacles.

While this solution allows characters to continue towards their desired goal, it does come with certain risks. For example, the waypoint network does not accurately describe the physical geometry of the environment. When a character moves away from the network, it runs the risk of falling off the geometry, becoming stuck, or entering various other dangerous circumstances (Tozour, 2008).

3.3.5   Road Network

The road network is the most complex type of navcontext, as well as the most important for use in a driving simulation. A road network is a more specialized version of a waypoint network, and is used to simulate roads that vehicles will move across. The

network itself has several components such as waypoints, intersections, and road segments. The road network object acts as a manager which controls these components (Presagis, 2011b).

Use of Waypoints in Road Networks

Similar to the waypoint network, the road network is made up of nodes placed at specific positions in the world which are connected together by edges. Whereas in a waypoint network these nodes are represented by waypoints, in a road network these nodes are represented both by waypoints, and by intersections. Waypoints are typically used when only two edges meet. They represent turns and bends in the road where a vehicle does not have a choice of which direction to take. This is especially useful to simulate winding roads, since the individual edges of the road network cannot be curved or bent.
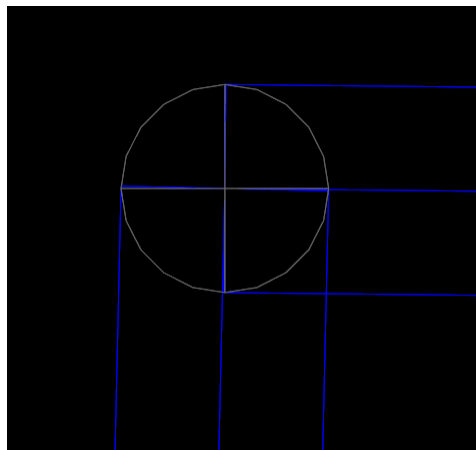

Figure 3.7: Waypoints are used to construct corners or bends in the road network that do not require the character to make a decision of which direction to take.

Road Intersections

Intersections are further specialized versions of waypoints. They are used to represent nodes where three or more edges meet, and a vehicle must make a decision on which edge to follow. An intersection is essentially a waypoint at its center, with new components build around it. One of these components which can be visualized is the stop line. The stop line represents the point on the network edge that is entering the intersection, at which characters must stop if the intersection is not safe to cross (for example, at a red light).

Even with these added components, the only unique attribute the intersection adds is the crosswalk width. This is a real number value which represents the width of the crosswalk, in metres, which lies between the waypoint at the center of the intersection, and the stop line. Increasing the crosswalk width causes the stop line to move further from the intersection's center. If this attribute is set to 0 metres, the crosswalk width will automatically be calculated based on the width of the incoming network edge.
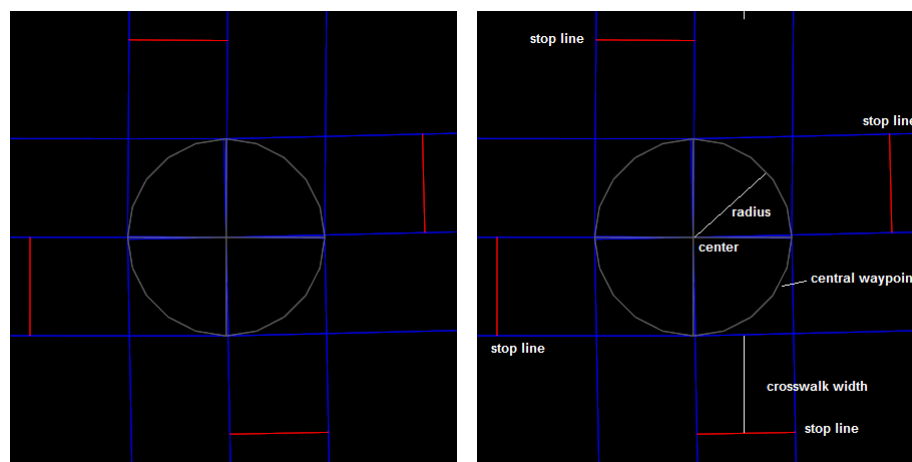
Figure 3.8: Intersections represent nodes where characters must choose one of multiple directions to take.

Road Segments

While the nodes of the network are represented by intersections and waypoints, the edges that connect them are represented by road segments. A road segment is a one directional cell connecting two nodes (either a waypoint or an intersection), which characters follow to move between the nodes.

Figure 3.9: A road segment is a single directional cell between two nodes. Characters using this road segment will move from the left waypoint to the right waypoint.

To create a simple two way road, two road segments are combined. One road segment moves from one waypoint to the second, while the other moves in the opposite direction. By connecting road segments together, a complete road network can be built.
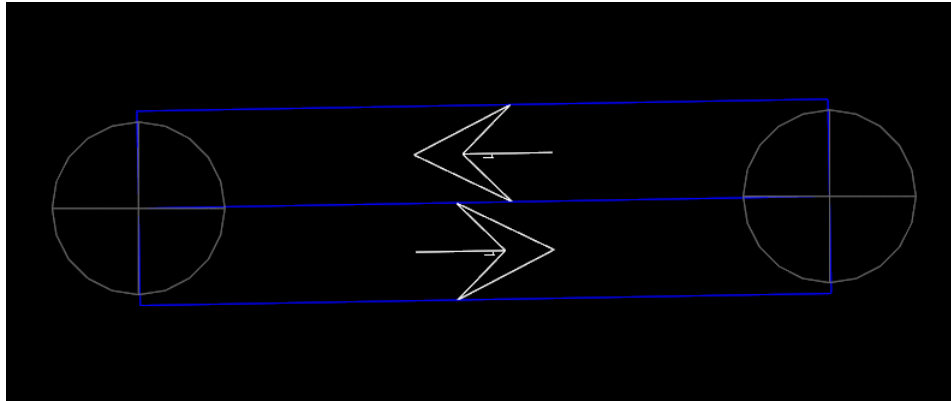
Figure 3.10: Two road segments connecting the same waypoints but in opposite directions can be combined to create a simple two-way road.

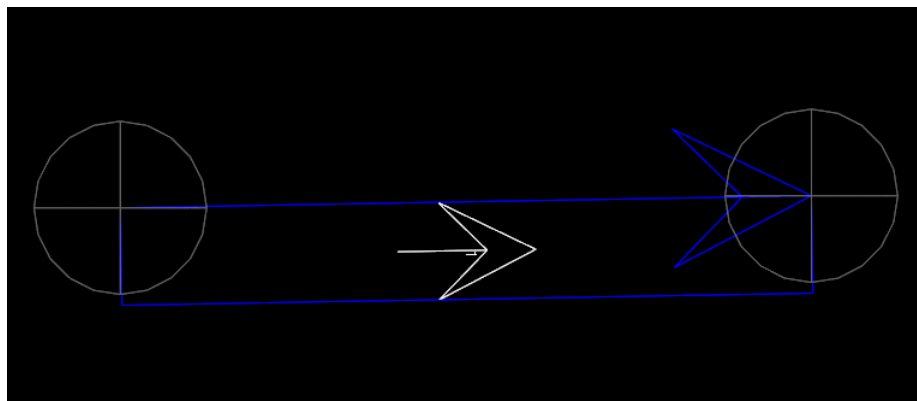Each segment in the network can be customized to fit any needed requirements. Each road segment contains attributes such as lane width, speed limit, and more, which can be modified to produce the desired road network.

Lanes

Road segments can be further divided into multiple lanes. Vehicles on the road network move through lanes as they move toward their target. The number and width of the lanes are stored as attributes in the road segment. Having multiple lanes allows multiple vehicles to share the same road segment without one vehicle having to remain stuck behind a slower moving vehicle. This eases congestion on the road network and allows vehicles to reach their targets more reliably. To further aid in vehicles' navigation, each lane is given an index, with the inner most lane on the segment being given the index 1 and the outer most lane having a lane index equal to the number of lanes on the segment. This allows the vehicle to easily identify which lane it is in, or to which lane it is moving.
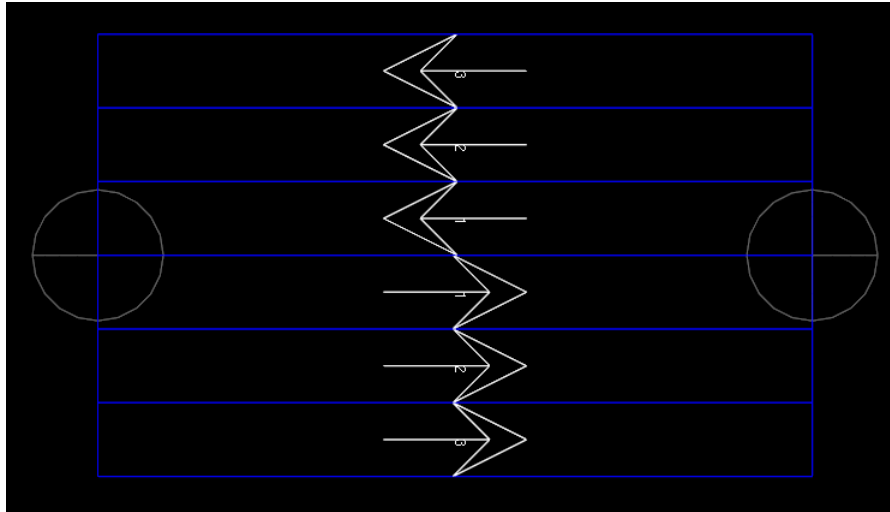
Figure 3.11: Road segments can have multiple lanes for vehicles to traverse. Vehicles will attempt to remain within one of these lanes.

Certain lanes can also be used as turning lanes. These lanes are used at intersections to direct vehicles that wish to turn into separate lanes from those that wish to proceed straight across the intersection. This helps control traffic flow as vehicles that are turning will move into turning lanes, preventing a queue of vehicles behind them.
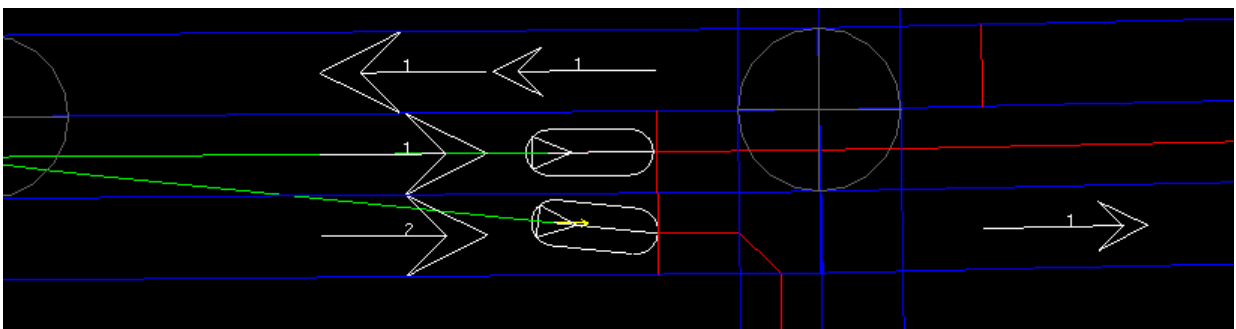
Figure 3.12: Turning lanes prevent vehicles intending to turn from blocking the path of other vehicles.

On multi-laned road segments, vehicles may have a lane they prefer to use over

the others. Which lane this is depends on the vehicle's driver type attribute. This attribute can be set to "fast", "medium", or "slow". A "fast" vehicle will always attempt to move into the lane with the lowest lane index. Conversely a "slow" vehicle will always attempt to move into the lane with the highest lane index. Vehicles with a "medium" driver type will not have a lane they prefer by default, and will choose which lane to use based on their current objective and surroundings.

Vehicles can also use different lanes for passing other vehicles. If a vehicle is stuck behind a slower moving vehicle, it can pass the slow vehicle using another lane. This is normally done by decrementing the vehicle's current lane index, which causes the vehicle to move one lane to the left (or one lane to the right if left-hand traffic is being used). The vehicle can then achieve its desired speed, then move back into its original lane after it has moved ahead of the slower vehicle.

This method of passing works well in many situations, but may fail if the vehicle attempting to pass is already in the lane with the lowest lane index. In this case, another method of passing can be utilized. Vehicles have an attribute called the frustration pass threshold. This attribute is a real number value which represents the level of frustration at which a vehicle will use a higher indexed lane to pass the obstruction in front of it. When the vehicle is forced to slow down due to some obstruction, that vehicle's frustration increases due to the vehicle being impeded from reaching its goal. Once the vehicle's frustration reaches the value of the frustration pass threshold, the vehicle will increment its current lane index, move to the higher indexed lane, pass the obstruction, and move back into its original lane. This method of passing helps ease congestion, particularly in environments with several different character types.

3.3.6   Traffic Indicators

Like their real world counterparts, traffic indicators control the flow of traffic at intersections to avoid collisions between vehicles moving in different directions. In the AI.implant, the two types of traffic indicators used are traffic signs, and traffic lights. These objects control the traffic flow by manipulating crossing conditions. The crossing condition is a value which dictates whether a character is permitted to enter the intersection, or must stop and wait. The traffic indicator maintains the current crossing condition for each of the lanes of the incoming road segments. If a crossing condition for a particular lane dictates that the character stop (due to a red light for example), the character will stop until the crossing condition changes.

Traffic Signs

Traffic signs are attached to the intersection. The type of sign depends on the road segment approaching the intersection. The road segment contains an attribute which specifies what type of sign will be used. The possible values for this attribute include "No Sign" (the default value), "Yield", and "Stop". When one of these values is selected, the sign object attached to the intersection will then conform to match the attribute. For the yield sign, vehicles will stop if there are any characters nearby that could obstruct their path and wait for them to move. For a stop sign, the vehicle will stop for a set amount of time before proceeding into the intersection.
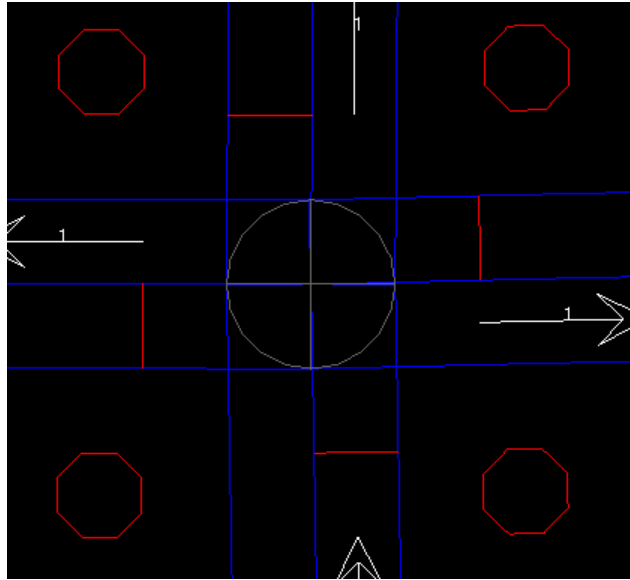
Figure 3.13: A sample intersection with stop signs present at each entry point.

Traffic Lights

Unlike traffic signs which remain static, traffic lights change their state at regular intervals. The traffic light object acts as a manager, and when attached to the intersection, it creates one light for each of the incoming road segments.
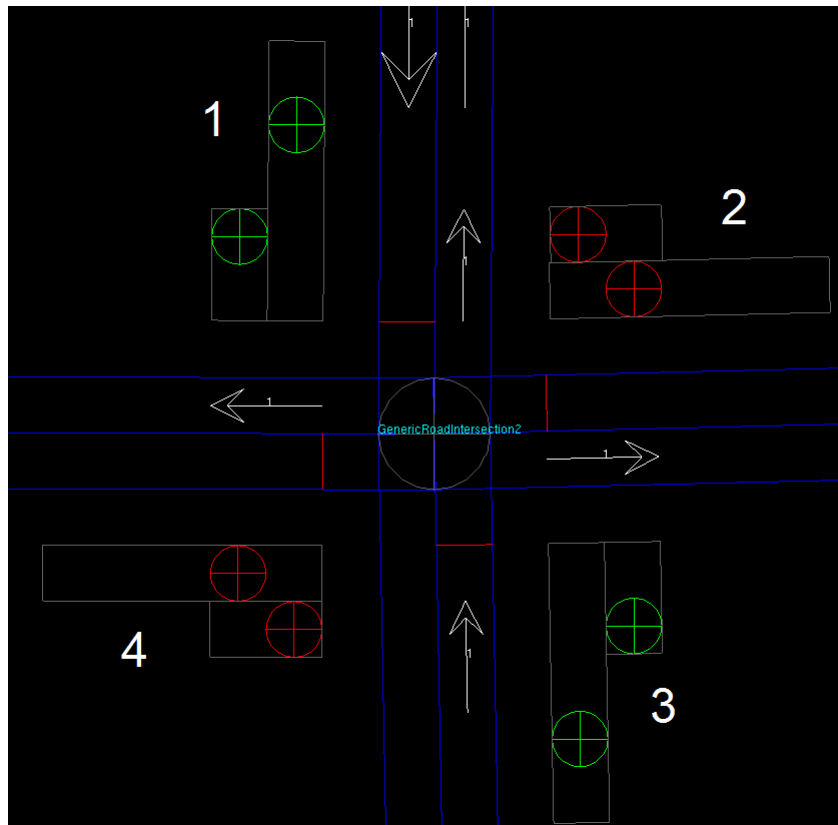
Figure 3.14: An intersection with traffic lights. One light is created for each incoming road segment.

The traffic lights cycle between red, yellow, and green states (the duration of which can be modified as required). The crossing conditions of the lanes at the intersection change depending on the current state of the traffic light. When a red light becomes green, the crossing condition for the lanes corresponding to the green light change to allow vehicles to proceed. Conversely the crossing condition for the lanes corresponding to the red light change to cause vehicles in those lanes to stop at the intersection's stop line. The traffic light may also have advanced turn lights, which manipulate crossing conditions for individual lanes, allowing turning lanes to proceed while others continue waiting.

3.4 Behaviours

Intelligence in the AI.implant is modelled with the simple "sense/think/do" paradigm of gathering environment data, applying the data to a set of pre-defined rules to make decisions, and performing the result of the decisions made (Kruszewski, 2006). Behaviours in the AI.implant are some of these sets of rules which describe how a character performs in the world. A behaviour is attached to a character, and will modify the actions of that character. Behaviours operate by calculating a steering force which moves the character, allowing it to become autonomous (Presagis, 2011b). Each unique behaviour represents a different action the character can undertake, and behaviours can be combined to produce even more complex actions. Additionally, behaviours contain several attributes which can be adjusted to fine tune the functionality of the behaviour for specific circumstances.

The behaviour operates by first outlining the goal the character must achieve, for example: move to this positon, or avoid this obstacle. Then, the behaviour calculates the speed, direction, and orientation the character requires in order to meet that objective. These values are collectively referred to as the character's "desired motion". The behaviour then generates the steering force based on that desired motion. The steering force will cause the character's current speed, direction, and orientation (or some combination of these, depending on the specific behaviour) to match its desired motion.

While the behaviour controls the actions of the character, it is also limited by the character it is attached to. Characters contain attributes which limit their movement, such as maximum speed and maximum acceleration. When a behaviour applies a steering force to a character, it cannot force the character to exceed any of these

bounds.

3.4.1   Combining Behaviours in the AI.implant

Each character can have multiple behaviours attached to it. Depending on the specific operation of those behaviours, they may be able to operate simultaneously without interfering with one another. Other behaviours however, may require additional attributes to prevent them from conflicting with one another.

To combine behaviours, the AI.implant utilizes two of the methods of behaviour blending outlined in (Reynolds, 1999). The first method used is to sum the steering forces calculated by each individual behaviour to find a net steering force. To assist in this, each behaviour has an intensity attribute, which acts as the weight for the summing calculation. Behaviours with a higher intensity will contribute more to the net steering force than behaviours with a lower intensity.

Alternatively, the AI.implant can use a priority system to manage multiple behaviours. Each behaviour contains a priority attribute, used to determine which behaviour will be active at any given time. When the priority method of blending is used, behaviours with a lower priority will be ignored whenever a behaviour with a higher priority is active. If a higher priority behaviour returns a zero vector as the steering force, the next highest priority behaviour is used instead, and so on.

The priority and intensity attributes ensure that behaviours will not interfere with each other by causing the contribution from some behaviours to be diminished, or removed entirely. Both of these blending methods have advantages and disadvantages, and the best choice depends greatly on the behaviours themselves.

3.4.2   Behaviour Types

Behaviours in the AI.implant are divided into sub-categories: simple behaviours, targeted behaviours, and group behaviours. Simple behaviours involve only the character they are directly attached to. Targeted behaviours involve at least two objects: the character they are attached to, and a target object or objects. Group behaviours involve groups of characters which act and move together, where each character in the group maintains the same speed and orientation with respect to the other group members (Presagis, 2011b). In all behaviour types, the character that the behaviour is attached to is referred to as the source character. The source character must be an autonomous character in order for the behaviour to function. For behaviours that have target objects however, the target can be an autonomous character, a non-autonomous character, a navcontext, a path, a group, a datum such as a position vector, or any other object which occupies a position in the world. Some behaviours though, are limited to targeting certain types of objects. For example, the target of a simple path following behaviour must be a path, whereas a target seeking behaviour can target characters, objects, or positions.

3.4.3   Avoidance Constraints

One of the key ways characters demonstrate intelligence outlined in (Kruszewski, 2006) is the ability to move through their world without hitting things. In the AI.implant, this is accomplished via the use of avoidance constraints. While avoidance constraints are not strictly behaviours in the context of the AI.implant, they still present a goal (avoiding collisions) and affect the movement of the character. Since they are not behaviours however, avoidance constraints are given priority over other

driving forces. Characters will attempt to avoid objects identified as obstacles even if it causes the execution of their behaviour(s) to become sub-optimal. Avoidance constraints are also similar to targeted behaviours in that they have a target or targets in addition to the character to which they are attached. These targets become the objects the character seeks to avoid.

In the AI.implant, avoidance of obstacles is a two-step process. First, threats of potential collisions must be assessed, and second, the character must react accordingly to the perceived threat (Kruszewski, 2006). Characters in the world contain a built-in radar which they use to detect potential obstacles, which have been identified as such by being the target of an avoidance constraint. The character then decides which obstacle poses the greatest threat of collision and moves to avoid it.
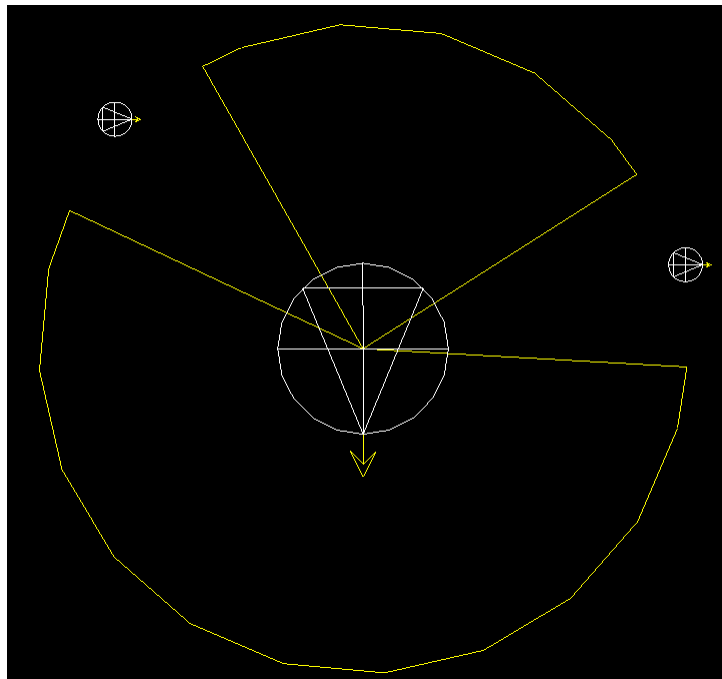

Figure 3.15: A character's obstacle detection service locating other characters it is attempting to avoid.

The AI.implant divides potential threats into four categories of collisions (Kruszewski, 2006):

1. **Stationary**: the character encounters a static, non-moving object in its path.

2. **Incoming**: an object is moving towards the character.

3. **Outgoing**: an object is moving away from the character, but the character is moving towards the object at a faster rate.

4. **Sideswiping**: an object will collide with the character from the side.

Additionally, the AI.implant defines two methods of avoiding these collision types: **circumvention** and **queuing** (Kruszewski, 2006). **Circumvention** is the act of avoiding the collision by attempting to move around the obstacle. This allows characters to avoid obstacles while delaying the execution of their behaviours by as little as possible. However, this method of avoidance can also introduce instability to character movement, as dodging obstacles may result in characters moving into areas they did not intend to.

Queuing is an alternative method of collision avoidance where characters slow down and potentially stop to wait until an obstacle has moved away from the character's intended path. This method of avoidance removes potentially unpredictable movement, but may be problematic in certain environments. Depending on the environment and the number of obstacles, a character may begin queuing and become stuck if many obstacles are present and are not cleared from the character's intended path.
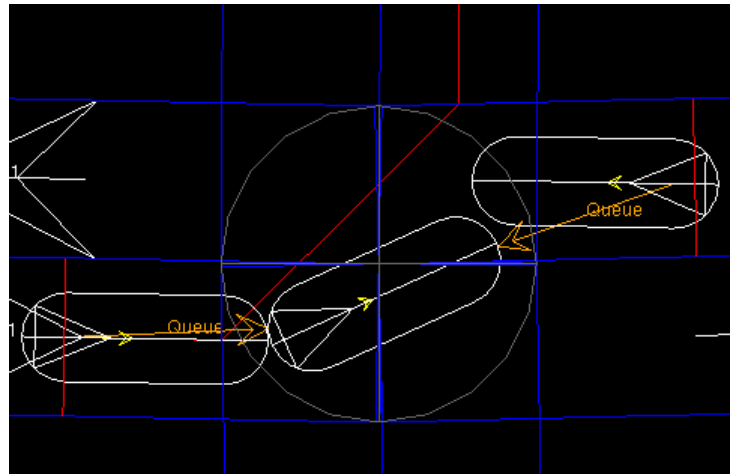
Figure 3.16: Depending on the different character types and behaviours, some characters may become stuck on each other while avoiding by queuing.

By utilizing different avoidance techniques paired with different collision types, different personalities of characters can be produced without modifying behaviour objects (Kruszewski, 2006). For example some characters may attempt to use queuing to avoid outgoing collisions, while others may be more impatient, and attempt to use circumvention to avoid these collisions.

3.4.4 Examples of Common Behaviours

While the AI.implant contains several behaviours for characters to utilize, some behaviours will be used much more frequently than others in the context of an urban driving simulation. For example, a simple milling or wandering behaviour is ideal for creating large crowds of pedestrians which move seemingly at random. Additionally, a simple target seeking behaviour provides a more direct way of controlling the motion of characters.

Since motion is one of the ways in which characters demonstrate intelligence

(Kruszewski, 2006), an understanding of how certain behaviours operate is critical to understanding the decisions and reactions of characters. This is best shown through the examination of some of the AI.implant's most useful behaviours.

Seek To

The Seek To behaviour is a targeted behaviour, meaning it involes multiple objects in its execution. Specifically, this behaviour has a source object (the object it is attached to) and a target object (the object the source will seek). The target of this behaviour can be any object with a position in the world, such as another character, a waypoint, or even a location specified in world coordinates.

This behaviour utilizes the attributes common to all behaviours, such as priority and intensity, as well as attributes common only to targeted behaviours. Additionally, the Seek To behaviour adds its own attributes which are more specific to its execution. These include:

- **Desired Speed**: the speed at which the source character will try to move to the target.

- **Contact Radius**: how far the source character must be from the target for the behaviour to consider the targeted as being reached.

- **Arrival Method**: how the character reacts to arriving at the target (character can be told to stop, or to continue moving past the target).

- **Slow Into Turns**: a boolean which controls whether or not a character will reduce its speed as it turns.

When the Seek To behaviour is active, it applies a steering force to move the character along a path to its target. The behaviour can also adjust the way the character moves to the target by modifying the offset, contact radius, arrival method, or other attributes. Once the character has reached the target, the character can stop, or begin seeking a new target depending on the specific implementation.

While seemily very simple, the Seek To behaviour is one of the most important behaviours in the AI.implant. This is because the AI.implant contains many other behaviours which include some form of target seeking as a method of controlling character motion, and as such, are specializations and refinements of Seek To. Because of this, the attributes of Seek To are reused by several other behaviours. These other behaviours include the ability to move a character towards a specified target, but also expand upon the Seek To behaviour to include more complex functionality. One example of this that is particularly useful in an urban traffic environment is the Wander behaviour.

Wander

The Wander behaviour has many similarities to the Seek To behaviour. It includes the functionality of Seek To, but adds new rule sets involving target selection. Like Seek To, Wander is a targeted behaviour, meaning it involves a source object and a target object. Unlike the Seek To behaviour however, the list of possible targets for Wander is much more limited. The target of a Wander behaviour must be a navcontext such as a navmesh or road network.

The behaviour simulates wandering by selecting a random point on the navcontext as the character's current target. The behaviour will cause the source character to

move along a path towards the selected destination point. When the source character reaches the target (or is within the contact radius), a new target point will be chosen, and the behaviour will steer the character towards it.

The Wander behaviour utilizes the attributes from Seek To, as well as its own attributes:

- **Minimum Wait Time**: the minimum amount of time the character must wait to select a new target after reaching its current destination.

- **Maximum Wait Time**: the maximum amount of time the character can wait to select a new target after reaching its current destination.

3.5 Paths

For many (though not all) targeted behaviours, the character's path through the environment needs to be at least partially calculated before the behaviour can steer the character. The character's path is the route it will take through its navcontext to the position its behaviour has identified as its destination. The path is composed of nodes stored in an array, with each node connected to both the previous node, and the next node in the path. When the character is told to move along its path, it will move to each of the calculated nodes in succession. The path is especially useful when the character's movement is restricted by a navcontext, such as a navmesh or road network, since the pathfinding calculations will take the character's movement constraints into account. The path nodes will be placed only along the cells that the character can navigate, helping to ensure the character remains on the cells it can move across. Once the path to the target has been calculated, the character will move
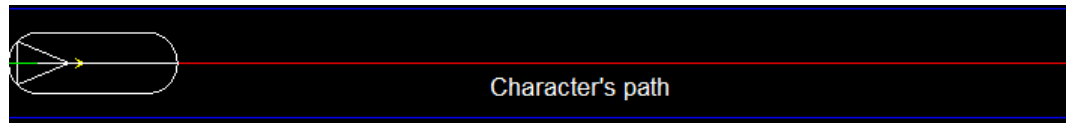
along that path in order to reach the target.

Figure 3.17: The character's path is calculated to provide it with a route to its target.

Each node in the path array contains several pieces of information. The node stores its position, the speed limit near the node, the indices of the cells entering and leaving the node, and the navcontext the node was created on. (Presagis, 2011a). If the node lies on a waypoint or intersection, it will also contain the ID of that object. These pieces of data are used to determine the location of the character relative to that node, and where the character is on its path.

The path itself can be in one of several states. Behaviours can take the current state of the path into account when calculating the steering force of the character. The current state of the path will also change depending on the current state of the target, character, and navcontext. The possible states of the path are (Presagis, 2011a):

- **Idle**: no path calculations are being done.

- **Unknown Source Position**: the source position of the path cannot be located.

- **Unknown Target Position**: the destination position of the path cannot be located.

- **Pathfinding**: the path is currently being calculated.

- **Have Partial Path**: calculations stopped with at least one path node calculated.

- **Have Entire Path**: calculations stopped with all possible nodes calculated.

- **Path Ends at Target**: the current path ends at the character's target.

Some behaviours however, do not require a path object, and instead can rely purely on the behaviour itself to achieve the desired character reaction. For example, a "Look At" behaviour only affects the orientation of the character, and as such does not require any pathfinding.

3.5.1   The Path Manager

Every character placed in the world has a path manager associated with it. The path manager is used to maintain the path of a character, and is responsible for the calculation of the path as well as aiding the character in navigating across the path. The path manager is also responsible for the multiple path refiners which act on the path, each with its own purpose.

3.5.2   Calculating the Path

The path manager obtains the character's current target from the character's behaviour. The path manager also stores the navcontext the character moves on, which allows the path manager to ensure that when the path is calculated it will not attempt to guide the character into an area it is constrained against moving into (Presagis, 2011a).

The path manager takes the location of the target and the current location of the character and uses them to calculate a path through the navcontext. The path manager uses a method of pathfinding referred to as "hierarchical pathfinding". Using

this technique, the path is first calculated in a low level of detail. This creates a generic path which can broadly guide the character, and then be refined into higher detail when needed. This is done to reduce the path finder's CPU and memory usage (Presagis, 2011b).

When the target is reached and a new target is presented, the path manager will update the current target location and advance the pathfinding calculation to find a new path.

3.5.3   Path Refiners

Path refiners are objects which are added to, and maintained by the path manager. They are used to make adjustments to the path based on certain conditions to help ensure navigation is optimal and correct. They refine the path into higher detail as the character navigates it.

When multiple refiners are used on the same path, the refiners are activated in the order they were added to the path manager. Each refiner modifies the path according to different criteria. For example, one path refiner may be used to smooth the path to eliminiate unnecessary nodes, while another refiner is used to help navigate around obstacles blocking the path.
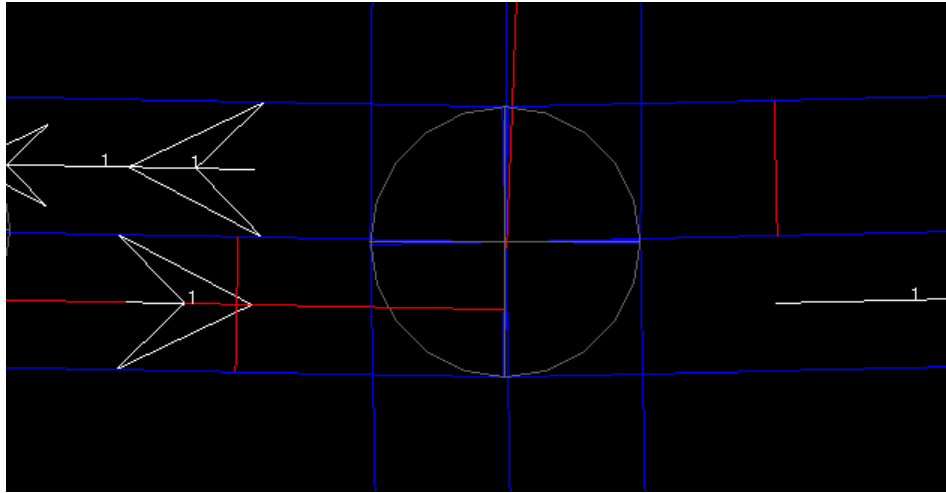
Figure 3.18: The character's path across an intersection prior to path refinement.
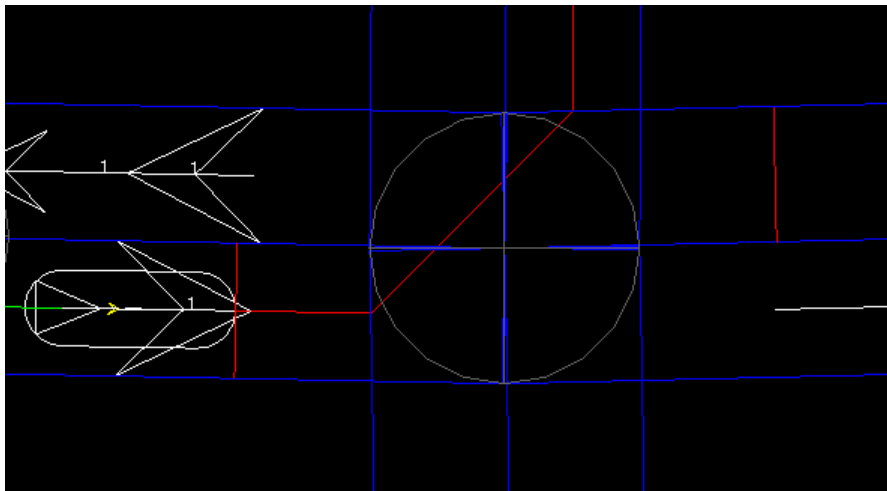
Figure 3.19: The character's path across an intersection after path refinement.

By modifying the path a character takes, path refiners are also capable of modifying the actions of a character without explicitly changing the character's behaviour. An example of this is the use of a path refiner which ensures characters respect traffic rules. This path refiner is responsible for ensuring characters stop at red traffic lights and stop signs. If the character begins approaching a red traffic light, the path refiner

will reduce the characters speed until it has stopped. Since the character's behaviour has not been changed, the character's target and goal remain the same. In this way path refiners can add new functionality to characters that can be used concurrently with their behaviours. The AI.implant includes four major path refiners managed by the path manager. Each path refiner makes adjustments to the path according to specific criteria.

Smoothing Path Refiner

The Smoothing Path Refiner handles the smoothing of the path. This path refiner may add or remove path nodes in order to ensure the character using the path moves smoothly and more predictably (Presagis, 2011a).

LOD Path Refiner

The LOD Path Refiner is responsible for the levels of details in a path. This refiner is used to specify the finer details of a path that was previously calculated more roughly by the path manager (Presagis, 2011a).

Obstacle Path Refiner

The Obstacle Path Refiner is used to adjust the path to guide the character around static or even semi-static obstacles in the world. This process will normally involve adding nodes to the path to bring the character away from the obstacle (Presagis, 2011a).

Traffic Path Refiner

The Traffic Path Refiner is used to adjust the path to respect traffic rules (Presagis, 2011a). This refiner is used to determine attributes such as the vehicle's speed on a road segment, when the vehicle must stop at a traffic light, or which lane the vehicle must be in, depending on its driver type. It then adjusts the path to ensure these requirements are met.

Additionally, there is a second traffic path refiner that is specifically used for pedestrian characters rather than vehicles. This path refiner will ensure pedestrian characters follow traffic rules at road crossings, but because pedestrians do not use road networks like vehicles, this path refiner does not include functionality to adjust certain vehicle-only parameters such as the current road lane.

3.6   Putting It All Together

Moving smoothly through the world is one of the fundamental ways in which characters demonstrate intelligence (Kruszewski, 2006). Each of these objects, from behaviours to path refiners, plays a crucial role in character motion. Behaviours are attached to characters to provide them with a target to move toward, the characters contain a path manager to determine a path through the environment, and the path manager contains multiple path refiners to improve the path as the character moves.
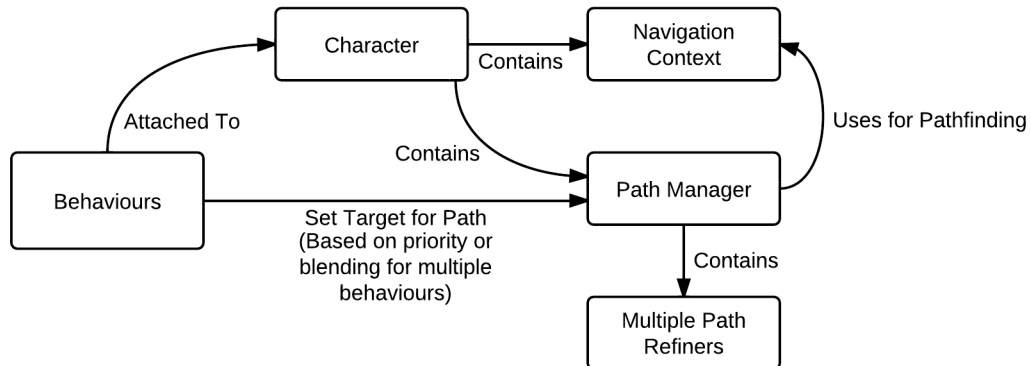
Figure 3.20: Several AI.implant objects are linked together to produce the desired character movements.

The manner in which these objects are used is as important as the objects themselves. The sequence of interactions that creates character motion needs to be understood to fully comprehend how characters interact with the world.

The primary instigator of movement of a character is the character's behaviour, as it is the behaviour that sets the target for the character to move toward (or away from depending on the behaviour). However, in order to reach the target in a realistic manner, the character needs more than just the target's location. The behaviour instructs the character's path manager to construct a path to the target. The path manager responds by constructing a path in a low level of detail first, to save on CPU and memory usage. The path manager uses the character's navcontext to ensure the path does not send the character where it cannot navigate.

Once the generic path has been constructed, the behaviour uses it to calculate the character's desired motion and will steer the character along that path. As the character progresses, the various path refiners in the path manager perform small modifications to each path edge to improve the level of detail. Both the calculation of the character's desired motion, and the refinement of the path edge occur every

frame to continue moving the character along the path.
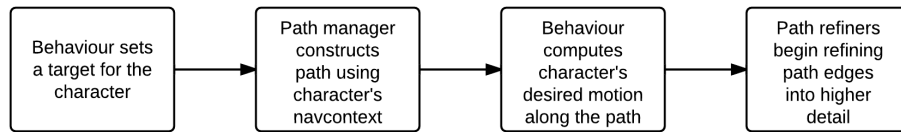
Figure 3.21: The sequence of events that occurs when a character moves through the world.

Some behaviours however, are much simpler than a target seeking behaviour and do not require an actual path. Because of this, these behaviours can skip straight from selecting a target, to computing the character's desired motion. These behaviours are typically either very situational and only used in specific circumstances, or do not require any change in the character's position (for example, a "Look At" behaviour only affects orientation).

Chapter 4

Custom Additions

As a component of the McMaster motion simulator, the AI.implant has been used to create the intelligent pedestrians and vehicles which populate the simulation. These characters offer the driver new ways of interacting with the simulation, and provide stimuli for the driver to respond to. However, we sought to create more potential driving scenarios in which the reactions of drivers could be examined, and to improve the appearance of autonomous characters to make the simulation more realistic, which causes drivers to react in the same way they would in the real world. To achieve this, new objects and routines were implemented in the AI.implant. New behaviours were added to allow characters to set new goals and make new decisions; new path refiners adapt a character's path for new situations, or allow a character to respond to the shape of its path in new ways; and new character types provide new types of interactions between the driver and the simulation.

4.1   Contributions

Signalling

Signalling refers to the use of lighting on vehicles to indicate a vehicle's intended motion to other characters. Signalling is split into two categories: brake lights and turning indicators. Brake lights are activated when a vehicle is reducing its speed, or is stationary, and have two possible states: "on" or "off". Turning indicators indicate the direction a vehicle intends to turn at an intersection, and can be in one of three possible states: "off", "left turn", or "right turn".

Emergency Vehicles

Emergency vehicles represent the real world vehicles which respond to emergency situations, such as ambulances and police cars. Vehicle characters must be able to switch between a regular motion strategy, and an emergency state, which has modified traffic rules for dealing with intersections. Additionally, an emergency vehicle must be able to utilize multiple lanes and road segments in order to avoid obstacles. Emergency vehicles also have priority on the road, causing other vehicles to yield the right of way.

Improved Turning Motion

For the default vehicle, making left turns at intersections causes the motion of the turn to appear segmented. The turning motion of vehicles needs to be improved so that it produces a smooth curve across the intersection.

4.2 Structure

The functionality of these additions is broken down into three new objects: A custom vehicle character, a custom path refiner, and a custom behaviour. These new additions interact with each other, and with existing AI.implant objects to fulfil the established requirements.

Signalling

Signalling is broken down into two categories: brake lights, and turning indicators. Each of these categories utilizes information on the vehicle's movement.

The functionality of brake lights relies only on the vehicle's current speed, and its intended speed. Therefore, brake lights have been incorporated into the custom vehicle character.

Turning lights however, rely on the vehicle's path to determine which direction the vehicle intends to move. Turning lights have been built into the custom path refiner, since the refiner examines the character's path and can extract information on the vehicle's intended direction.

Emergency Vehicles

The functionality of emergency vehicles is broken down into several smaller components.

A mechanism which switches a vehicle from normal traffic rules to emergency traffic rules, and vice versa, is needed to control which vehicles act as emergency vehicles. This mechanism has been implemented as an attribute in the custom vehicle, allowing each vehicle's emergency state to be controlled separately.

The emergency traffic rules themselves also need to be added, since emergency vehicles behave differently on road networks than default vehicles do. The traffic rules which vehicles obey are enforced by their path refiners. Because of this, the new traffic rules have been implemented as part of the custom path refiner.

Additionally emergency vehicles require new logic regarding lane changes, since they may use other lanes or other road segments to pass obstructions. This new lane changing logic has been implemented directly into the vehicle itself, as the vehicle is responsible for tracking which lane it is in, and can check the lane ahead of it for obstacles.

Other characters in the world also need to react to emergency vehicles. This reaction has been implemented as a new behaviour. This enables all characters to react correctly in the presence of emergency vehicles.

Refined Vehicle Turning

The refinement of vehicle turning takes place during left turns at intersections. The path refiner examines the path and knows when a vehicle intends to turn left at an intersection. Because of this, the refinements have been incorporated into the custom path refiner.
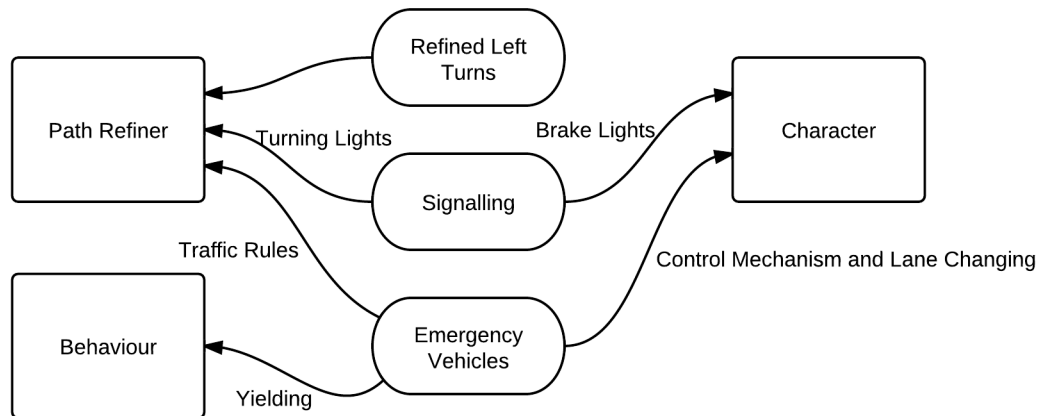
Figure 4.1: The different custom components added to the AI.implant and how they are distributed among different AI.implant objects.

4.3   Signalling

While the default vehicle in the AI.implant offers an accurate recreation of an automobile, it is still missing some components that would improve the vehicle's appearance in the simulation if added. An example of this is the use of signal lights on vehicles. The use of signals, such as brake lights and turning indicators, is not only important to improving the appearance of vehicles in the simulation, but also affects the quality of the human driver's response to the autonomous vehicles they interact with.

4.3.1   Brake Lights

Since brake lights are easily recognizable components of any automotive vehicle, it is important to capture their functionality in order for the simulation to appear realistic. Additionally, brake lights on autonomous vehicles may be used as a stimulus when examining a driver's behaviour when on the road. Brake lights have been implemented

in the custom vehicle character.

To control brake lights, the custom vehicle character contains a state variable called "braking". This state variable is used to store the current state of the brake lights, which can either be active or inactive. When the value of this state variable is changed, the vehicle sends a message to the scenario manager, which then informs the image generator that the state of the brake lights has changed. The IG then updates the 3D model of the vehicle on the screen.

The vehicle uses a simple algorithm to determine when brake lights need to be applied. The vehicle stores the speed at which it wishes to move, according to its behaviour, as its desired speed. Additionally, the vehicle stores the speed it is presently moving at as its current speed. If the vehicle's desired speed is lower than its current speed, it indicates the character is slowing down. If that is the case, the brake lights are activated. Additionally, if the vehicle's desired speed is 0 m/s, the vehicle is stopped, or in the process of stopping, and the brake lights are activated. Otherwise, the brake lights are disabled.

4.3.2   Turning Indication

Turning indicators are used to signal the intended motion of drivers at points where they have multiple possible motion paths available, such as intersections, and the driver must make a decision on which to take. The addition of turning indicators not only improves the appearance of vehicle characters, but also creates new possible driving scenarios with which to examine drivers' reactions to the movements of other vehicles. Because of this, we felt it necessary to add this functionality to our simulation.

Turning indicators must be present on all vehicles in the simulation. Since the indicators are used to show the vehicle's direction of travel in advance, the correct indicator must be activated before the vehicle makes its turn. To achieve this, the turning indicators have been incorporated into the custom path refiner. Since path refiners examine the path of the vehicle, the custom path refiner is able to identify turns in the vehicle's path, and determines which turning indicator it needs to activate.

The algorithm the path refiner uses to determine which traffic indicators to activate is summaraized as follows:

1. Determine if the vehicle is close to an intersection.

2. Determine if the vehicle is turning.

3. If the vehicle is turning, determine the direction of the turn (left or right).

4. Activate the corresponding turning indicator.

5. Once the turn is complete, deactivate any turning indicators.

The refiner determines if the vehicle is close to an intersection by examining the node in the vehicle's path that it is approaching. The path node has an attribute which stores the ID of the waypoint or intersection it lies on (this attribute is set to zero if the node is not on a waypoint or intersection). If the node lies on a waypoint, the refiner does not need to activate any turning indicators, since waypoints are used to represent bends in the road. If the vehicle is approaching an intersection, however, the refiner must calculate the distance between the intersection and the vehicle. The refiner only activates turning indicators when the vehicle is within 50 metres of the intersection. This is necessary because the refiner may find a turn in the vehicle's

path that is still far from the vehicle. Activating a turning indicator early may be confusing to the human driver.

Next the refiner must determine if the vehicle intends to turn at the intersection it is approaching. The refiner does this by examining the road network cells that are approaching and leaving the path node (the indices of these cells are stored as attributes in the path node). The refiner determines the direction vectors of these two cells, and calculates their dot product. The value of the dot product corresponds to the angle between these two vectors. Using this proportionality, the path refiner defines a range of values which indicate a turn in the path. If the dot product of the two vectors falls within this range, the refiner knows the path turns.
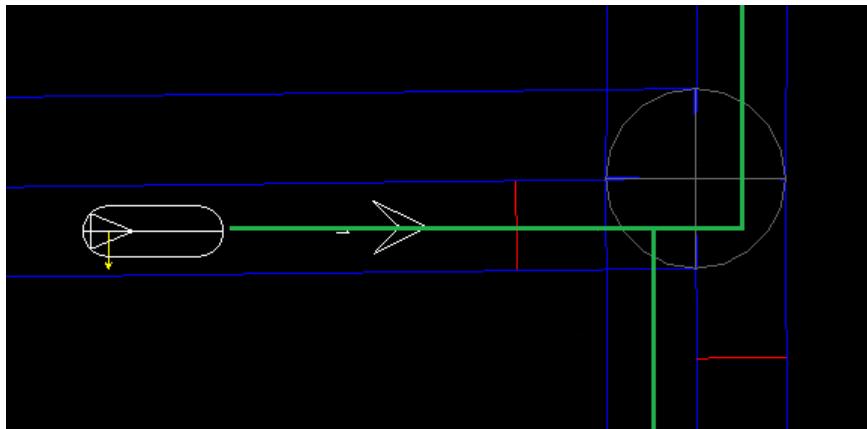

Figure 4.2: The possible turn decisions a vehicle could make at an intersection. The angles are calculated from the dot product of the direction vectors.

Next, the refiner must determine the direction the vehicle intends to turn. The refiner takes the cross product of the two direction vectors that were used previously. Due to the nature of the cross product, the result of this calculation will be a new vector perpendicular to the two direction vectors. This new vector's z component will be either a positive floating point value, or a negative floating point value. Due to

the right-hand rule, which specifies the direction of the cross product of two vectors, a positive z component indicates the vehicle's path is turning left, while a negative z component indicates the vehicle's path is turning right. Once the direction of the turn has been found, it is encapsulated with the ID of the vehicle in a message, which is sent to the scenario manager. The scenario manager then updates the image generator, which activates the correct turning light on the 3D model of the corresponding vehicle.

In addition to determining when the vehicle is turning, the path refiner must also determine when the vehicle has finished its turn so that it disables any active indicators. There are a number of ways for the path refiner to determine when it should deactivate any turning indicators. Firstly, if the object the vehicle is approaching is a simple waypoint rather than an intersection, then the vehicle is approaching the meeting point of two road segments and will not have to make a decision regarding which direction to take. Additionally, if the distance between the vehicle and the next intersection is too great, it indicates the intersection is too far away for any turning indicator to be active. Lastly, if the dot product of the two direction vectors of the road cells does not fall within the range that would indicate a turn, the vehicle is not turning. If any of these conditions is true, the refiner sends a message to the scenario manager to disable any active turning indicators.

4.4   Emergency Vehicles

Emergency vehicles are the specialized characters used to represent the real world vehicles that are designated to respond to emergency situations. For example, police cars, ambulances, and fire trucks are all recognized as emergency vehicles and are marked as such by the use of sirens and specialized vehicle lighting. These vehicles
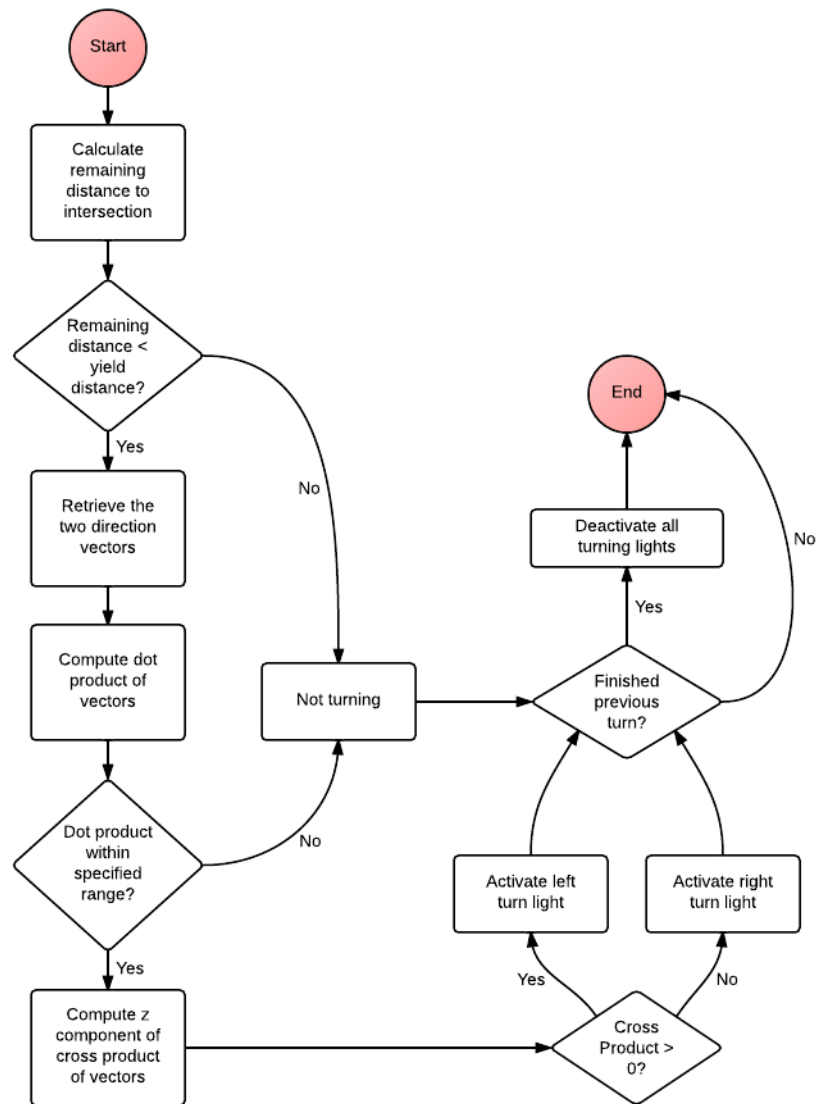
Figure 4.3: The algorithm used by the path refiner to determine which turning indicator to activate.

are permitted to disregard conventional traffic rules for the sake of reaching their destination quicker, and are given priority on the road by all other vehicles.

The functionality of emergency vehicles is divided between the custom vehicle character, the custom path refiner, and a new behaviour for vehicles and pedestrians. When in use, emergency vehicles have a new set of traffic rules specific to them. Additionally, other vehicles on the road must yield priority to the emergency vehicles, so they may reach their target's location more quickly.

4.4.1   Toggling the Emergency State

Ironically, a key feature of emergency vehicles is that they do not always behave as emergency vehicles. In the real world, when an ambulance switches off its lights and sirens, it behaves as any other vehicle on the road does. Therefore, an emergency vehicle in the AI.implant must be able to also act as a normal vehicle which obeys conventional traffic rules.

To achieve this, there must be a mechanism to switch between the emergency vehicle traffic rules, and normal vehicle traffic rules. This mechanism has been implemented as a new attribute on the vehicles themselves. This "emergency" attribute is present on every one of the custom vehicles in the simulation, so any of these vehicles may act as an emergency vehicle (though in practice, only certain pre-selected vehicles will be used as emergency vehicles).

4.4.2   Dealing with Traffic Indicators

Emergency vehicles respond to traffic indicators differently than vehicles using conventional traffic rules. In the real world, emergency vehicles have the right of way at

intersections and may proceed through them regardless of any traffic lights or signs. In order to properly convey this priority, the path refiner takes steps to modify the responses of emergency vehicles to certain crossing conditions.

The refiner finds the index of the lane the vehicle is using to approach the intersection. With this index, the refiner determines the crossing condition of the lane the vehicle is in. Rather than modifying the crossing condition value for a particular lane, the refiner changes how the emergency vehicle responds to certain crossing condition values. When a vehicle is in its emergency state, and it encounters a crossing condition instructing it to yield, the vehicle will instead ignore that condition and maintain its speed through the intersection, trusting other vehicles to yield to it. Similarly, when a vehicle in its emergency state encounters a traffic indicator with a crossing condition instructing it to stop, the vehicle will instead reduce its current speed until it is moving at 15% of its maximum speed, and proceed through the intersection. This allows emergency vehicles to proceed through intersections that normal vehicles could not, just as real world emergency vehicles do.

4.4.3   Lane Changes and Passing

A challenge that arose during implementation involved the way in which emergency vehicles handle lane changes. In the real world, emergency vehicles always attempt to move to open lanes to avoid other vehicles. If there is no available open lane and the road ahead is blocked, the emergency vehicle will move to the opposite side of the road to move past the obstruction. In the AI.implant however, vehicles attempting to circumvent obstructions in front of them risk temporarily leaving the road network. This is one of the disadvantages of using waypoint networks to navigate

(Tozour, 2008), and road networks share this weakness. Since the space next to the road network is used as a sidewalk for pedestrians, emergency vehicles need to be prevented from leaving the road network.
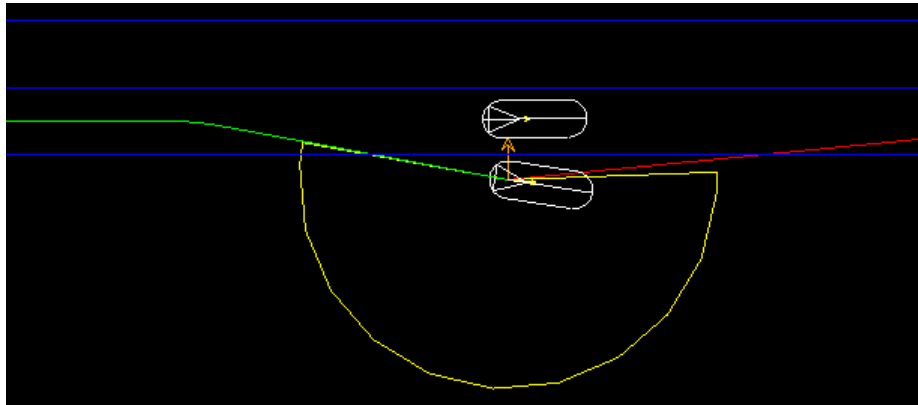

Figure 4.4: Attempting to dodge obstructions may cause the vehicle to leave the road network.

To alleviate this issue, new functionality has been added to the custom vehicle, which helps emergency vehicles handle changing lanes and passing other vehicles. Each vehicle is capable of looking for obstacles in a lane on its road segment by creating a search capsule, centered at its position, and extending in front of, and behind the vehicle, at specified distances. When the vehicle is in an emergency state, it checks the lane ahead of it looking for any obstructions within 25 metres. If an obstruction is found, the vehicle must find a new lane. Using simple lane selecting logic, the vehicle chooses a new lane index, and moves into that lane.

However, if an open lane is not found, the vehicle must cross to another road segment to avoid the obstruction. The vehicle examines the road network to find the road segment opposite the one it is on. The vehicle will move into that segment to pass any other vehicles in its way. While the vehicle is using the opposite road segment to pass, its speed is reduced to 25% of its maximum. As the vehicle is

passing, it continues to check the road ahead for obstacles. Once the original road segment is free of hazards, the vehicle returns to this segment and resumes moving at normal speed.

Finding a New Lane

In order to facilitate avoidance of obstructions via lane changes, the vehicle needs to select a new lane to move into. This new lane is selected by using a simple algorithm to examine all the possible lanes on the vehicle's road segment. The algorithm functions as follows:

1. Check for obstructions one lane to the left, and one lane to the right.

2. If an open lane is found, move into it.

3. If no open lane is found, check for obstructions two lanes to the left, and two lanes to the right.

4. Continue in this manner until either an open lane is found, or until every lane has been checked.

In order to look for obstructions, the vehicle checks for obstacles in its lane anywhere within 25 metres in front of it. This procedure is also used to find an open lane, as lanes which do not have obstructions within 25 metres are used to avoid lanes which are obstructed.

The vehicle systematically searches for an open lane by utilizing the indices of the lanes on the road network. The vehicle first uses the index of the lane it is in. Using this index, the vehicle is able to search one lane to the left (by subtracting one from

the index) and one lane to the right (by adding one to the index). If a lane is found
to be free of obstructions, the vehicle selects it as the new lane, and moves into it.
If neither of the lanes are open, the vehicle then searches two lanes to the left and
right, and so on. The vehicle continues searching in the same manner until either an
open lane is found, or the algorithm has searched all the lanes of the road segment
and has not found an open lane. If no lane was found, the vehicle will then attempt
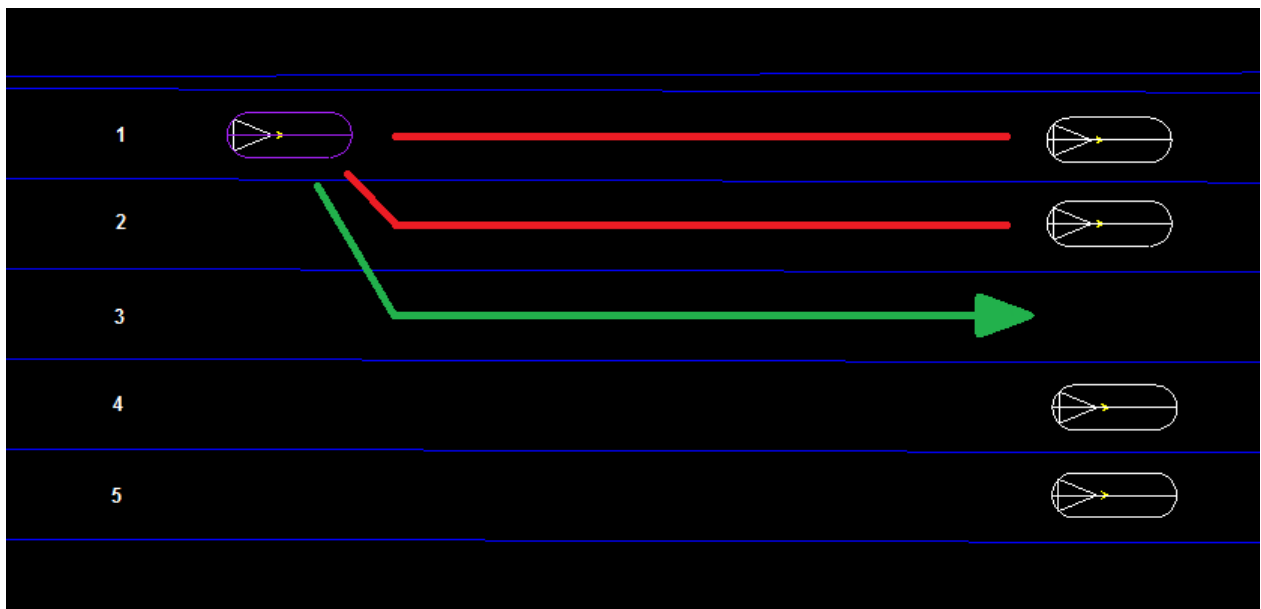to use the opposite road segment for passing.

Figure 4.5: The emergency vehicle checks lanes to its left and right (if they exist)
until it finds an open lane it can use.

4.4.4   Yielding to Emergency Vehicles

In order for emergency vehicles to accurately portray their real world counterparts,
more functionality is needed than just the vehicles themselves. When a real world
emergency vehicle moves down a road, other vehicles in its path are obligated to
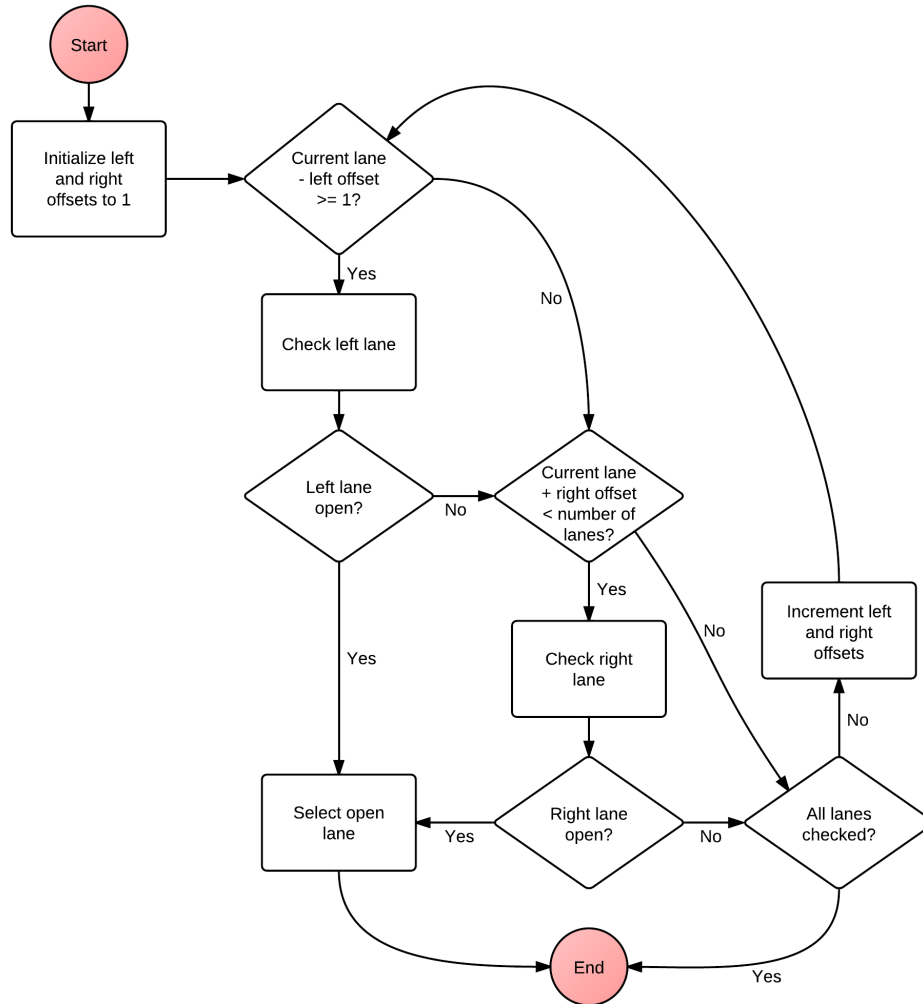
Figure 4.6: The algorithm the vehicle uses to find an open lane.

reduce their speed and pull over to ensure the emergency vehicle can safely pass. In the simulation, this is accomplished with the use of a new behaviour which reacts to the presence of emergency vehicles. This "Yield To" behaviour will cause characters to yield the right of way to emergency vehicles.

Yield To is a targeted behaviour, meaning it is dependent on at least two objects. The source object is the character the behaviour is attached to, whose motion is being modified. The target or targets are the vehicles the source character must yield to. Since there may exist multiple emergency vehicles on the same road network, the Yield To behaviour targets a group object. A group is a collection of objects in the world. When a group is targeted by a behaviour, the behaviour treats all members of that group as a target. Groups can have multiple members, a single member, or no members, making it ideal for a situation where the exact number of emergency vehicles is unknown, or can change as time progresses. The behaviour targets a group, and the emergency vehicles are added to that group. Vehicles are added to or removed from the group as their emergency state changes, allowing vehicles to transition in and out of their emergency state without having to update all other characters in the world.

In addition to inheriting the attributes common among all behaviours and targeted behaviours, the Yield To behaviour also adds its own unique attributes:

- **Yield distance**: the distance from the target at which the source character will begin yielding.

- **Yield to oncoming**: a boolean that controls whether vehicles yield to targets even if they are on the road segment opposite the one the vehicle is using.

The first task the behaviour must accomplish, is to establish which vehicles the

source character is attempting to yield to. Since the target is a group that may contain multiple members, the behaviour must decide which member of that group to examine. The behaviour first ensures the group it is targeting has members. If the group is empty, the behaviour has no potential targets to yield to. If the group does contain members, the behaviour will use the source character's position to determine which group member is the closest to the source character. This closest group member then becomes the local target of the yielding.

Once the target has been established, the behaviour calculates the distance between the target and the source character, and compares it to the value stored in the behaviour's "yield distance" attribute. If the distance between the target and source is greater than the value of the yield distance, then the target is too far away for the behaviour to consider. Since the target has already been established as the closest emergency vehicle to the source character, the behaviour determines that the source vehicle does not need to yield.

While the Yield To behaviour may be applied to all characters, from this point onward the behaviour's functionality diverges. The behaviour functions differently depending on the type of character to which it is attached. Specifically, the behaviour operates differently for vehicles than it does for pedestrians.

Yield Behaviour for Vehicles

The goals for vehicles using the Yield To behaviour are simple:

1. Avoid moving into the path of incoming emergency vehicles.

2. If already in their path, move away if possible.

If the source vehicle has already stopped (at a red light for example), the behaviour keeps the vehicle still, to prevent it from moving into the path of the emergency vehicle. If the source vehicle is moving however, the behaviour will instruct the vehicle to stop, and attempt to move it away from the emergency vehicle's path as it slows. The behaviour gradually reduces the vehicle's speed by lowering its desired speed. This weakens the steering force acting on the vehicle, lowering its speed. As the vehicle slows, the behaviour may attempt to move the vehicle towards the edge of the road network, to create more space for the emergency vehicle. If the vehicle is in one of the inner lanes of the road network, it does not have sufficient space to safely move aside. However if the vehicle is in the outermost lane of the road network, it is instructed to pull over onto the curb to allow the emergency vehicle to move past.

To do this, the behaviour retrieves the source vehicle's left vector, which is a vector extending from the left side of the vehicle that is used to help determine the vehicle's orientation. The behaviour uses this vector to determine the direction the vehicle must move in order to pull over. On a road network using left-hand traffic, the left vector is used unmodified. However in right-hand traffic, the left vector is first reversed to provide a new vector extending from the right side of the vehicle. This is done by reversing the numeric sign on the x and y components of the vector (the z component is near zero). The behaviour then uses the right vector as the desired direction of the source vehicle, and creates a steering force to guide the vehicle in that direction.
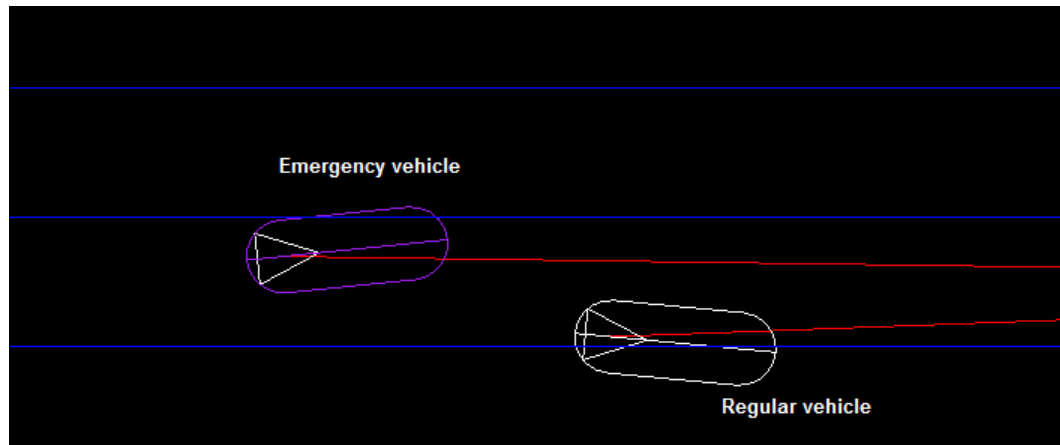
Figure 4.7: The normal vehicle pulls to the side of the road to accomodate the emergency vehicle.

Although the new steering force affects the direction the vehicle is moving, it does not affect the orientation of the vehicle. This is to cause the vehicle to move to the side of the road, while still facing forward. If the behaviour affects both the vehicle's direction and orientation, it produces a motion similar to turning, causing the character to face away from the road network. This makes moving back onto the road network after the emergency vehicle has passed more difficult, as the vehicle must then move across a sidewalk to turn back onto the road network.

The vehicle continues slowing until it has stopped completely. It then waits until distance between it and the emergency vehicle is greater than the yield distance. Once this occurs, the vehicle resumes moving towards its target.
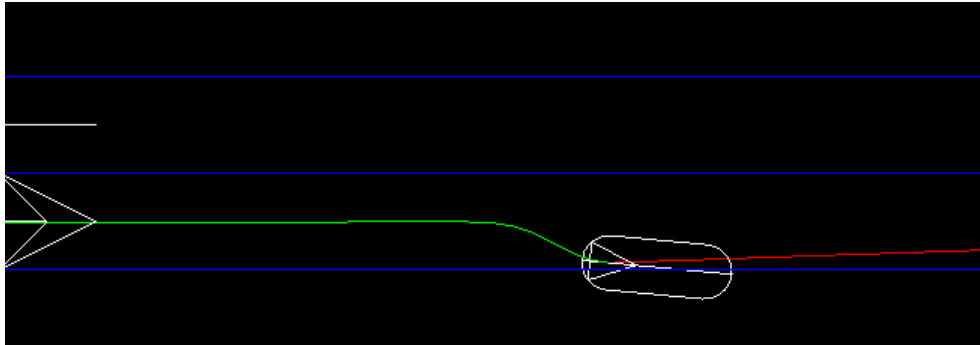
Figure 4.8: When the steering force only affects direction, the vehicle remains facing the direction of travel on the road network.
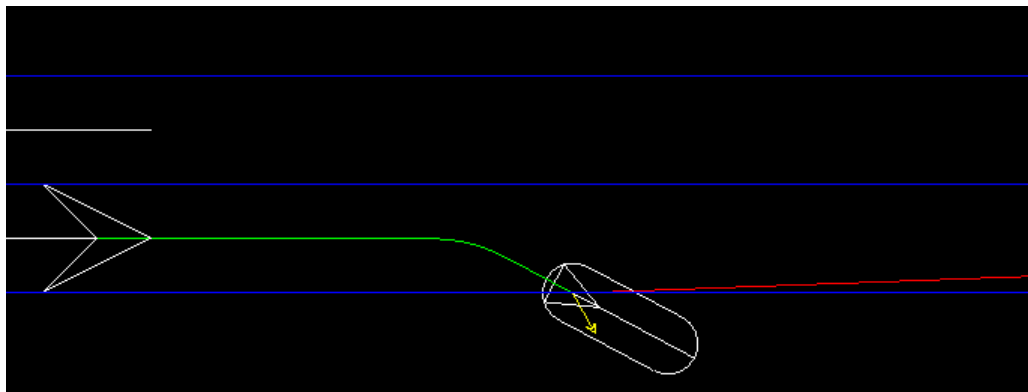

Figure 4.9: When the steering force affects both direction and orientation, the vehicle's resulting movement causes it to turn much further off the road network.

Yield Behaviour for Pedestrians

Since pedestrians do not use the road network to navigate, they do not rely on the same algorithm used by the vehicles when yielding to an emergency vehicle. However, pedestrians are capable of crossing over road networks at specific points by using crosswalks. Because of this, pedestrians are still capable of influencing the motion of vehicles on the road.

The overall goals of the yield behaviour for pedestrians are similar to those for

vehicles:

1. Avoid moving into the path of incoming emergency vehicles.

2. If already in their path, move away if possible.

The behaviour achieves these goals by finding the index of the pedestrian's current cell on its navmesh. The cell is used to determine the position of the pedestrian relative to the emergency vehicle that is being targeted. Since the cells which are navigable for the pedestrian only intersect the road network at certain points, the behaviour is able to determine whether or not the pedestrian is currently obstructing the path of the emergency vehicle. The behaviour examines the pedestrian's current cell on the navmesh for blind data. Specifically, the behaviour looks for the string value used by pedestrians as the crosswalk tag. If the tag is found, the pedestrian is on a crosswalk and is blocking the emergency vehicle's path. If this is the case, the behaviour accelerates the pedestrian to maximum speed until it has left the crosswalk. This is to simulate the act of a pedestrian running to allow the emergency vehicle to proceed through the intersection as quickly as possible.

Additionally, the pedestrian's current cell is used to prevent it from moving into the path of an emergency vehicle. The behaviour examines the regular blind data of the cell and determines the type of surface the pedestrian is on. If the pedestrian is on a cell which does not represent a crosswalk, its speed is reduced to 0 m/s to prevent it from stepping onto a crosswalk, and into the path of the emergency vehicle.

4.5   Refinement of Left Turns for Vehicles

The way in which vehicles turn at intersections presented another challenge which needed to be overcome. Vehicles proceed through intersections differently than they do through normal waypoints when turning. In the AI.implant, each intersection has specific entry and exit points depending on the positions of the lanes moving into, and out of, that intersection. The vehicle's path refiners place path nodes at these entry and exit positions to guide the vehicle through the intersection. While this successfully guides the vehicle into the correct position at the end of the turn, this method of controlling a vehicle's turns may adversely affect the quality of the vehicle's motion.
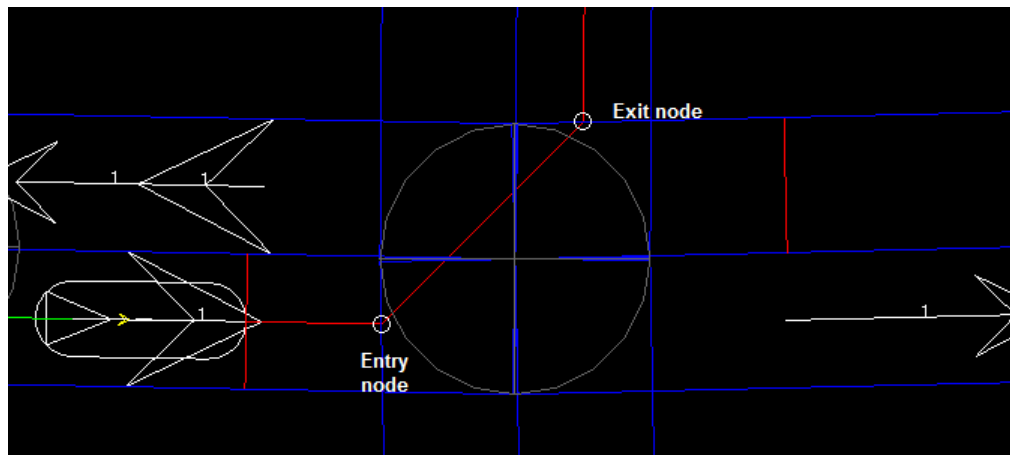
Figure 4.10: The default path refiners create new path nodes to allow a better turning motion through intersections.

When the vehicle attempts to turn left at an intersection, it reaches the entry node, turns towards the exit node, reaches the exit node, then turns to match the direction of the new lane. While this achieves the desired effect of turning into the correct lane, the turn itself becomes visibly broken down into smaller discrete turns,

rather than one continuous curve. This effect becomes pronounced in the simulation, where the image generator represents the vehicles with 3D models.
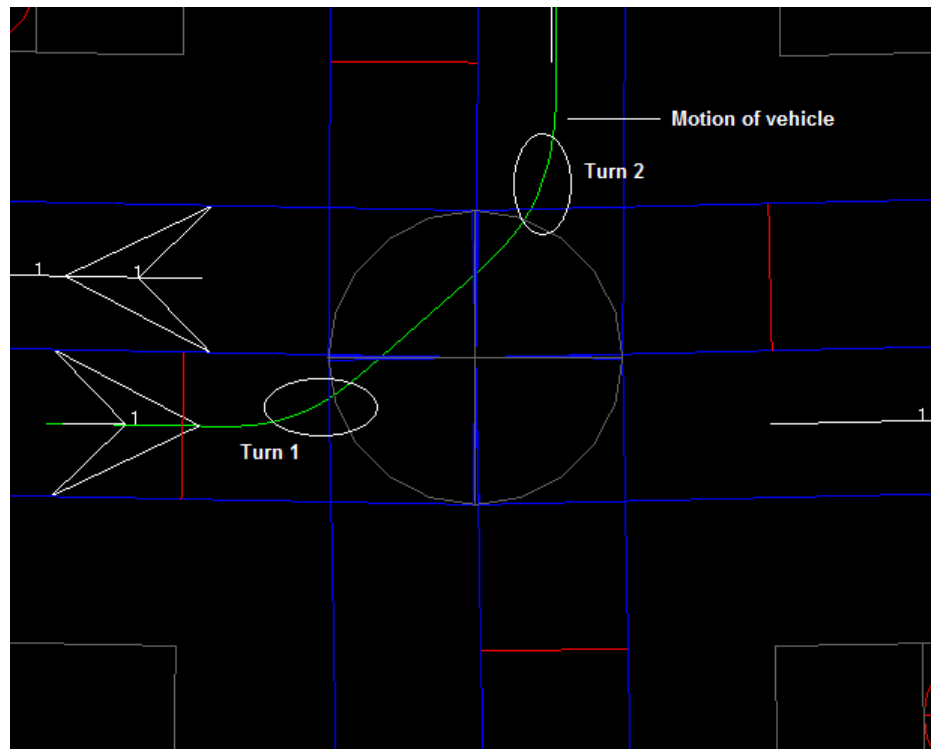

Figure 4.11: As a result of the multiple path nodes, the character's motion becomes broken down into two separate turns.

The vehicle's path refiners can determine when the vehicle is attempting to make a left turn through an intersection because of the way they examine the vehicle's path. This allows the path refiners to make adjustments to the way the vehicles turn at intersections to improve performance. Rather than modifying the vehicle's path itself, the refiner modifies the vehicle's attributes causing its turning to become smoother. Modifying attributes is a much simpler method of controlling a vehicle's movement than attempting to add new path nodes, or changing the positions of existing path nodes.

Vehicles contain attributes called "max left turn" and "max right turn". These attributes represent the maximum angular speeds at which the vehicle can make left and right turns. The default values for these attributes are high enough that when the vehicle enters an intersection, it turns quickly enough to finish changing its orientation before it has reached the exit point of the intersection. This causes the turn to appear broken down into smaller turns.

To address this, the path refiner reduces the maximum left turning speed of the vehicle, while leaving the maximum right turning speed unmodified (since modifying this value could adversely affect the quality of right turns, which do not need refining). To achieves this, the refiner converts the vehicle's linear speed into angular speed, by using the size of the intersection as the radius of the vehicle's turning curve. This new angular speed is used as the maximum left turning speed of the vehicle. With this new turning speed, the vehicle finishs turning as it reaches the exit point of the intersection, produceing a much smoother looking turn.
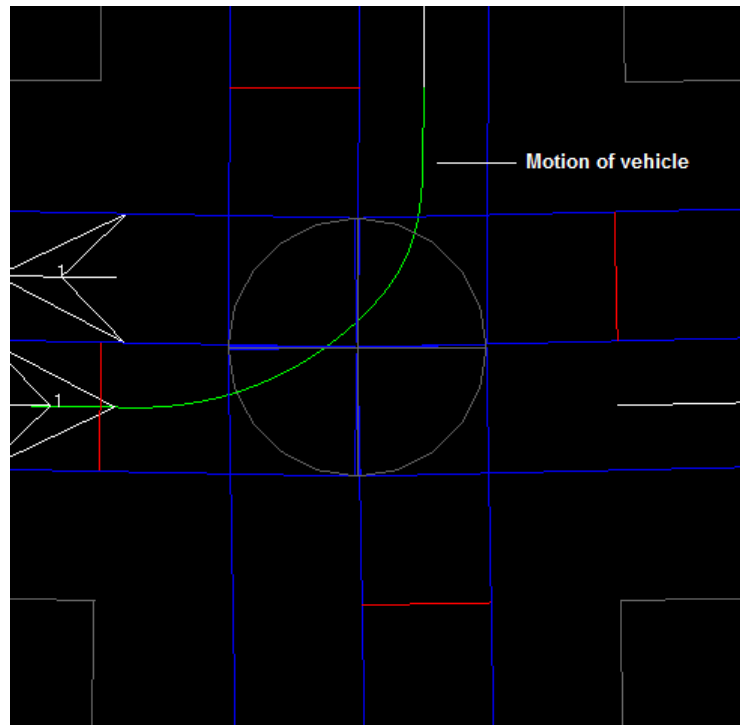
Figure 4.12: The left turn of the vehicle appears much more continuous when its turning speed is reduced.

Once the vehicle has exited the intersection, its maximum left turning speed attribute is returned to its default value. This is because vehicles move around different objects in different ways. For example, path refiners do not place entry and exit nodes when the vehicle is crossing a waypoint. In modifying the vehicle's attributes to better accomodate one road feature, the vehicle's performance around a different road feature may be adversely affected. Returning the max left turn attribute to its default value avoids this risk.

Refining Turns for Emergency Vehicles

There does exist one exception for which the vehicle's path refiner will not adjust the value of the max left turn attribute for a left turn at an intersection. This condition is

if the vehicle is currently in a passing state, meaning it is attempting to move around another vehicle. This is done to better accomodate emergency vehicles. Because emergency vehicles have the right of way on road networks, when approaching an intersection the emergency vehicle will attempt to pass any vehicles waiting for the traffic light to change. Because of this, the emergency vehicle may not always be entering the intersection at the correct entry point. Attempting to modify the turn rate of this vehicle at such a time may prevent it from reaching the intersection's correct exit point. If the emergency vehicle is not attempting to pass another vehicle, however, it will be entering the intersection at the correct location, meaning the max left turn attribute may be safely decreased.

4.6   Strategies of Characters

The setup of the custom characters, behaviours, and path refiners, and how they work together is just as crucial to the simulation as the components themselves. While a character's behaviour is responsible for setting the character's desired goal and the character's method of reaching that goal, characters can still be influenced by a variety of other components. For example, path refiners enforce traffic rules on characters regardless of their behaviour. A character's overall motion strategy is dependent on its behaviours, its path, its path refiners, its avoidance strategies, and even the character itself. These components need to be combined in the correct way in order to produce the desired appearance of characters in the simulation. Characters need to be assigned the proper behaviours, their attributes need to be determined, and their navigation contexts need to be selected. Only when every component of a character's motion strategy is determined will that character perform as required.

Use of Groups

Group objects are useful for organizing multiple characters to improve behaviour performance. Characters that share the same type or have similar attributes can be placed in a group together to ensure that behaviours treat each member in the same manner. For example, by using groups, pedestrians will be able to avoid all vehicles, regardless of small differences in attributes between individual vehicles.

Our setup of group objects in the simulation utilizes three groups, which represent the possible motion strategies of different characters:

1. **Pedestrian Group**: contains all pedestrian characters.

2. **Vehicle Group**: contains all vehicle characters. Characters in this group may be moved to the Emergency Group if their emergency state is active.

3. **Emergency Group**: contains all vehicles which currently have an active emergency state. Characters are added and removed from to this group according to their emergency state.

4.6.1   Pedestrian Strategies

Pedestrians are represented by the basic Person character. They represent individual humans which move along sidewalks and crosswalks. In the simulation they are seen on sidewalks next to the road network, utilizing crosswalks to move between sections of sidewalk. They appear to simply wander through the world acting as obstacles for the driver and other vehicles. Pedestrians will also attempt to avoid other characters using different avoidance strategies for different character types.

Navigation

Pedestrians use a navigation mesh as their navigation context, and move between cells to reach their objectives. Pedestrians also make use of blind data on the navmesh to determine which cells are navigable to them. Pedestrians contain a pathfinding constraint, which limits their movement to cells which have a blind data value within a certain range. Cells representing sidewalks and crosswalks have blind data values within this range, while cells representing roads do not. Pedestrians also make use of the crosswalk tag, to determine where they are crossing road networks and must obey traffic rules.
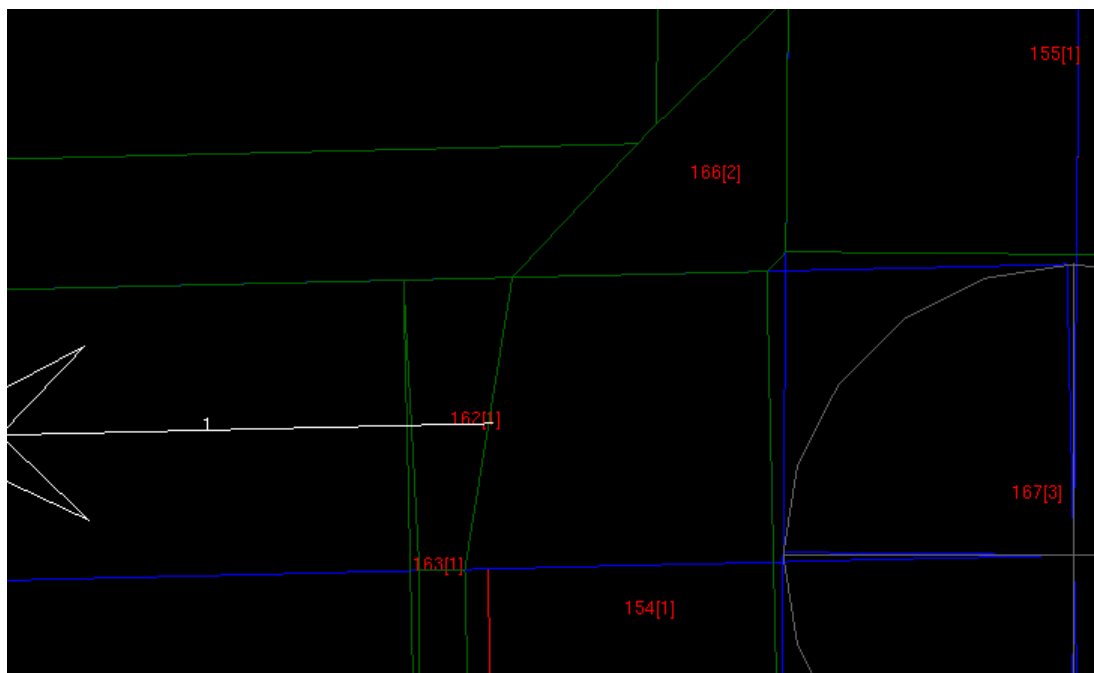

Figure 4.13: A sample section of navmesh on which a pedestrian would navigate. Cell IDs are shown on the cells with their corresponding blind data value in brackets. A blind data value of 1 indicates a crosswalk, 2 indicates a sidewalk, and 3 indicates a road.

Behaviours

Pedestrians have two behaviours which are combined using a priority system:

1. **Yield To**: pedestrians will react in accordance with a Yield To behaviour when approached by vehicles belonging to the emergency group. The Yield To behaviour is given a higher priority than the other behaviours associated with the pedestrian.

2. **Wander**: pedestrians use a wander behaviour which targets the navmesh. The behaviour chooses a random point on the navmesh (possible target positions are limited to cells navigable by pedestrians) which the pedestrian will move towards. The pedestrian's path manager constructs a path to the target, which is further improved by the path refiners. When reached, the behaviour selects a new target location.

Avoidance

Proper avoidance constraints are key to preventing characters from colliding with one another. However, each character type reacts to every other character type in a different way. For pedestrians, the two types of characters they must avoid are other pedestrians, and vehicles. Avoidance of emergency vehicles is handled by the Yield To behaviour.

When avoiding other pedestrians (members of the pedestrian group), pedestrians will avoid by queuing, meaning they will wait until the character blocking their current path has moved, rather than attempting to move around it.

Conversely, pedestrians will avoid vehicles by attempting to circumvent them while maintaining a clearance of 2 metres. This clearance is to ensure the pedestrians

maintain a comfortable space between them and the vehicles.

4.6.2   Vehicle Strategies

Vehicles are represented by the custom vehicle character. They include all the functionality of the AI.implant's default vehicles, as well as the custom additions used for signalling and lane changing for emergency vehicles. Additionally, they use the custom path refiner which further aids in signalling, smooths the vehicle's path around left turns, and establishes emergency traffic rules. In the simulation, they will appear as vehicles moving along a road with potentially multiple lanes of traffic. They make directional decisions at intersections and respect traffic indicators. Vehicles also queue behind other vehicles and pedestrians as an avoidance strategy to remove the risk of collisions.

Navigation

Vehicles use a road network as their navigation context. They move between waypoints and intersections via road segments. Vehicles tend to remain in their preferred lane (depending on their driver type attribute) unless they are instructed by their behaviour to move to another. Vehicles will also respect traffic rules and any traffic indicators on the network.

A critical problem that arose for vehicle navigation involved dead-ends on the road network. If the network stopped abruptly, vehicles could reach the end waypoint and become unable to continue forward. This prevented the vehicle from reaching its next target, and could cause traffic jams if more vehicles queued behind the vehicle that

was trapped. This problem was overcome by activating an attribute on the dead-ending road segment which allowed U-turns. This allowed vehicles which reached the end to turn around and use the opposite road segment to return in the direction they came.

Behaviours

Vehicles contain the same behaviours as pedestrians, though with slightly different attributes for each:

1. **Yield To**: vehicles will yield to members of the emergency group. This behaviour has priority over the vehicle's other behaviours, but is disabled if the vehicle enters an emergency state.

2. **Wander**: vehicles will use a wander behaviour which targets the road network. Like with pedestrians, the behaviour selects a point on the network that the vehicle's path manager will construct a path toward. When reached, the behaviour selects a new target location.

A behavioural issue that arose with the use of a wander behaviour for vehicles, occured when the target location was within an intersection. Often the vehicle would reach the target but, as a result, be slightly inside the bounds of the intersection. The vehicle would then move through the intersection regardless of the current state of any traffic indicators. This was resolved by increasing the contact radius attribute of the behaviour. This meant the behaviour would consider the target as being reached when the vehicle was still a few metres away, and would select a new target before the vehicle entered the intersection.

Avoidance

Vehicles must avoid the same groups as pedestrians, but do so in different ways.

When avoiding pedestrians, vehicles maintain a clearance of 2 metres and avoid by queuing. Vehicles will stop 2 metres away from pedestrians and wait until their path is no longer obstructed.

Similarly, vehicles also avoid other vehicles by queuing. This is necessary to maintain steady traffic flow, particularly at intersections with traffic indicators. Vehicles maintain a clearance of 1 metre from each other while queuing. Avoidance of emergency vehicles is once again handled by the yield to behaviour.

When a vehicle is being used as an emergency vehicle, it reacts to other characters in much the same way, with the exception that it will not queue behind vehicles. Vehicles with an active emergency state will instead attempt to avoid other vehicles via circumvention.

Chapter 5

Conclusion

The custom built vehicle character, path refiner, and behaviour help produce the illusion of intelligence in autonomous characters, which both improves the quality of the simulation by making it appear more realistic, and by adding new ways for the driver to interact with the simulation. The addition of signal lights makes the vehicles the driver interacts with appear more like real world vehicles; the refinement of vehicle attributes for left turns at intersections improves the appearance of motion for vehicles, which in turn makes them appear more realistic; and the inclusion of emergency vehicles introduces characters the driver associates with urban environments, and adds new scenarios in which the responses of drivers can be examined. These additions have improved the McMaster motion simulator, and helped achieve the goal of creating a simulation with which to examine the reactions of drivers in customized driving scenarios.

The AI.implant has been effective at creating and managing the autonomous characters present in our system. Our success in utilizing an artificial intelligence to improve the quality of our simulation demonstrates the usefulness of AI systems in

designing simulations. As the use of simulators rises, further research into artificial intelligences will certainly benefit the simulation industry, making simulations more robust, and more realistic.

Bibliography

Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, **11**(1), 125–130.

Goodwin, S., Menon, S., and Price, R. (2011). Pathfinding in open terrain. School of Computer Science, University of Windsor.

Kruszewski, P. A. (2006). A game-based cots system for simulating intelligent 3d agents. Presagis Canada Inc.

Lester, P. (2005). A* pathfinding for beginners. `http://www.policyalmanac.org/games/aStarTutorial.htm`.

Presagis (2011a). *AI.implant SDK Documentation*. Presagis Canada Inc.

Presagis (2011b). *AI.implant User's Guide*. Presagis Canada Inc.

Reynolds, C. W. (1999). Steering behaviours for autonomous characters. In *Game Developers Conference Proceedings*, pages 763–782. Miller Freeman Game Group.

Tozour, P. (2008). Fixing pathfinding once and for all. `http://www.ai-blog.net/archives/000152.html`.