University of Dublin

Trinity College

Sketch-based Path Control

by

Brendan Carroll, B.A.Mod Computer Science

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science

University of Dublin, Trinity College

September 2010

Declaration

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

Brendan Carroll

September 13, 2010

Permission to Lend and/or Copy

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

Brendan Carroll

September 13, 2010

Acknowledgments

I'd like to thank my supervisor John Dingliana for his tireless efforts to help me and to my friends and family for supporting me through the development of this thesis.

Brendan Carroll

University of Dublin, Trinity College
September 2010

Sketch-based Path Control

Brendan Carroll
University of Dublin, Trinity College, 2010

Supervisor: John Dingliana

Sketch-based interfaces are largely confined to experimental research and are used in specialist areas such as those occupied by artists, architects and engineers. While mouse driven input and menu based interfaces have become the common computer interaction method, sketch input is an intuitive, natural way of communicating intent and ideas. Pathfinding algorithms are a very important part of modern interactive entertainment applications. They can be used in a multitude of data structures and research into their operation continues to this day. Interaction with these algorithms has been primarily through the use of mice which limit the amount the user can contribute to their function. This project implements an approach for the use of sketch-based interfaces with pathfinding so that it is possible to control and modify the paths of agents on 3D terrain using sketch strokes in real-time.

List of Figures

Contents

1 Introduction

1.1 Aims

The aims of this project are to develop a system which uses a sketch-based interface to interact with agents in a 3D environment through the use of sketch strokes. With this interface, it will be possible to select an arbitrary number of agents and sketch particular paths they should take over the terrain itself. The project should be able to condition strokes for their use as paths, allow the modification of agents paths using sketch, and do all of this in real-time.

1.2 Motivation

Interest has been growing in the last few years in non-mouse based interfaces with massive investments being made in multi-touch surfaces and motion based input devices. Even with the rise of such devices, sketch-based interfaces receive little attention compared to others. While stylus based interactive entertainment devices are common, most resort to point and click mechanics and no functionality that explicitly uses sketch strokes is employed. As this is the case, this project aims to use a sketch-based interface to control agents in a 3D world by allowing the creation and modification of paths through terrain, all in real-time. It is hoped that by demonstrating this, more sketch-based devices will use sketch strokes as a viable way of interacting with agents in a 3D scenario.

1.3 Outline

This thesis is split into two main parts. The first part covers the background which contains the theory and techniques of areas implemented in this project. The second part contains the design, implementation, evaluation and conclusion of the project based on the techniques laid out in the background. Each of the two will be briefly explained.

The background contains two chapters, one dedicated to sketch-based interaction(Chapter 2) and the other to navigation(Chapter 3). Sketch-based interaction is described, starting with a brief outline of the area and the benefits and negatives of using sketch as an input tool. It then explains the major research areas today in sketch-based interaction. These areas are detailed with current papers and approaches being described. Relevant information to the field which does not fit into these research areas are then explained and the summary describes an overview of the field as a whole. The navigation chapter outlines the current algorithms and approaches to navigation in modern applications. Different graph data structures also called world representations are explained with their positives and negatives being mentioned. Pathfinding algorithms are described with the most common in use today being mentioned and heuristic algorithms currently in use also being explained.

Chapter 4 details the overall design of the system developed here. The different components of the project and how each of them work and interact is also explained.

Chapter 5 describes the tools used to construct this project and the different parts which needed to be created for the end result. Pathfinding algorithms used are explained as is the sketch input system and how the input points are used for stroke creation. Then the analysis of the sketch strokes, so gestures the user gives can be recognised and actions executed are described. The graphical rendering component is also explained giving information about what visual information had to be added to the project.

Chapter 6 contains information on evaluating the success of the project.

The conclusion chapter(7) contains the conclusion to the project but also details information on possible future work for it. Numerous improvements are explained and possible directions for furthering the project itself.

Part I

Background

2
Sketch-based Interaction

This chapter contains information on sketch-based interaction. Background information is described with the most common and current methods available for approaching certain areas being mentioned.

2.1 Outline

Sketch-based interfaces take data from the user in the form of strokes. The user draws strokes onto a tablet or some type of touch responsive surface, using a pen-like stylus. Most interfaces can take in the pen's position on the surface while others can additionally take the pressure at the point it is being pressed, the angle it is being held and how close it is to the surface should it not be touching it at all. Through all of this, sketch interfaces try to emulate the use of a pen on paper and apply it to the use of a computer. Sketching is a natural way of communicating, conveying ideas and with a few simple strokes, complicated and abstract concepts can be expressed. Sketching is a form of drawing and is used in a multitude of fields. It is used in planning stages for designers and architects, conveying complicated principles, abstract ideas and in art for preparatory work or the finished piece itself.

It is hoped that by using sketch as the primary interface to computing, it will make the operation of computers more accessible and naturalistic. Recent trends over the years have been towards finding new ways of accepting input from the user, such examples include the motion sensing Wii remote[35],

Microsoft Kinect[31] and the popularity of multi-touch devices. Research into sketch-based interfaces and modelling has also increased. There is now a yearly workshop on the topic, hosted by Eurographics and submissions have been plentiful[12]. The following sections describe the current status and applications of sketch-based interfaces.

2.2 Benefits of Sketch-based Interaction

There are many benefits to sketching and sketch-based interfaces. Sketch-based interaction is an intuitive, simple communication method. The physical activity and mental processes associated with sketching form the basis for problem solving, development and general creative thought for most of the areas where 3D modelling is used. As sketching is often used in the early stages of development in the creative process when details are rough and designs are vague, this area is the exact stage of 3D modelling which is under-served by the current market[10].

With regard to model creation, sketch-based variations have shown themselves to be significantly faster than traditional WIMP(Windows, Icon, Menu, Pointing Device) for creating 3D models[28].

2.3 Problems with Sketch-based Interaction

There are numerous problems with sketch-based interaction. It has several critical aspects, due to difficulties during the interpretation step by the computer. These mainly derive from the semantic gap between the users communicative intention and how he/she is able to convey it[5]. Solving ambiguity is an ongoing problem with several different approaches being suggested to address it, such as analysis of the context in which the command is issued[27], or relying on the knowledge of the user's drawing style[2].

Sketch-based interaction can be intuitive if designed carefully but the idea that the user should instinctively know what to do in most situations is false as many gesture based systems still have to be given cognitive effort and practice from the user to remember and master commands. WIMP based systems already demand this level of effort so sketch-based interaction will not be placing any increasing strain on users than what is already being asked of them today.

Modern WIMP based interfaces are largely incompatible with sketch-based controls as they are primarily based on the point and click actions provided by a mouse. For sketch-based interfaces to become common, a shift would have to occur away from WIMP based interfaces.

2.4 Model Creation

Model creation is the subfield of sketch-based development with the largest amount of research devoted towards the topic. It involves reconstructing a 3D model based on 2D input provided by a sketch device. Olsen et al[37][36] divided the range of model creation techniques into two categories which will be used here to explain current developments. The categories are the Evocative and Constructive. Evocative creation systems use sketch to instantiate a built-in model which is most similar to the input, which means it has a pre-computed model stored and when a sketch is drawn, it picks the model which it thinks the sketch is closest to. A constructive creation system maps the input strokes to a model itself, the strokes are used to create geometry which has not been pre-computed.

2.4.1 Evocative

In the SKETCH system[49], the user draws strokes which are used to divide 3D primitive objects such as cones, cylinders, spheres, objects of revolution, prisms, extrusions, ducts and superquadrics. The strokes are also used to place the object in the scene. Template creation systems are a common way of interpreting strokes. Using this system, the user would draw a series of strokes, and a collection of pre-computed 2D templates would be checked until the most common one was found. Then the 3D shape which is contained in that template would be inserted into the application. Template systems can allow the user to create simple or complex shapes depending on the corresponding templates and the strokes usually have to have a reasonable degree of resemblance.

Yang et al[49] used a template based system to create complex 3D geometry. Their system also created the resulting 3D object using measurements of the user-defined input strokes and were able to create models of planes, mugs and fish.

The benefits of such template systems are their extensibility: new recognitions can be added to the applications as fast as the template can be created, although such complex shapes can also limit them in their applications as they are more specific.

Shin and Igarashi propose a system which they called the "Magic Canvas System"[40]. This approach uses templates to create scenes themselves. When the system takes in user sketch input, if a corresponding shape is found in the templates then those objects are placed in the scene according to that input. The object is also rotated and scaled to match that input and the relationships between objects is also inferred with objects possibly being placed on top of each other, e.g. a cup on a book.

Lee and Funkhouser[24] propose a hybrid template system. The more complex shapes would be retrieved from the database, but the objects would all be parts of complex models. The user would then

join all of these objects by sketching the connections. Lee and Funkhouser say "it allows creation of highly detailed models/scenes (as details come from parts in the database), while 2D sketched strokes provide all the information for part selection and composition (no 3D manipulation is required, in general)." They report the application as a success with users finding scene position very easy with sketch strokes. They did have 907 shapes in their template database so many shapes which a user wanted to draw had a decent probability of being similar to a template in their database.

2.4.2 Constructive

Constructive model creation takes in sketch input from the user and without using any matching templates, tries to reconstruct a 3D object based on the strokes alone. This is much more difficult than template based methods as there is more ambiguity determining what the user is trying to do. Constructive model creation is a very difficult problem, one which covers many different disciplines, because of this there are many different methods for constructing different types of geometry.

There are a number of techniques which try to determine the meaning behind each stroke that is drawn. Line labeling is an algorithm which classifies lines as contour, concave or convex edges. Extensions on this idea come as systems try to label corners, edges and vertices of objects in the hope of reconstructing the geometry. The strokes can be labelled as the user is drawing them but are only reconstructed once input has stopped[42].

One method for solving ambiguity poised by sketch input is to apply previous knowledge attained, a form of machine learning. Lipson and Shpitalni[26] propose an approach whereby a 3D model is reconstructed using the 2D input sketched but taking into account models it has encountered in the past. They "demonstrate how a simple correlation system that is exposed to many object-sketch pairs eventually learns to perform the inverse projection successfully for unseen objects."

Another method involves multi-view systems. In these systems, additional strokes can be added to define more constraints to the surface and from other viewpoints. This added information allows developers to gain more information about the users intentions and reduces ambiguity. Nealan et al's FiberMesh system[34] allows the user to draw additional strokes onto a surface and changing viewpoint to modify strokes already drawn.

2.4.3 Alterations

Augmentation is the process of adding something to a mesh already created. In Teddy[19], the user must draw a contour line to define the area of the model where the appendage is to be fixed. The

user would then draw some strokes indicating the shape of the new limb. FiberMesh[34] also allowed for augmentation but at a much simpler level. A stroke could be drawn on the surface of a mesh and the user could then displace that stroke, thereby altering the surface of the mesh. These alterations would only be superficial with creases being created either protruding or extending into the surface.

Numerous methods for the deformation of meshes exist with sketch-based implementations. Cutting, bending, twisting, tunnelling and free-form deformations are some of the alterations that can be made. Sketch-based deformations like augmentations, are usually straightforward to use as the model being operated on has already been created. Some systems already mentioned[34][19] allow handles which the user can select and drag thereby altering the stroke and the model that the handle is connected to. When the stroke is moved, it deformed orthogonal to the viewing plane.

Kara et al.[20] propose a deformation system based on templates. The user draws some strokes, the system then selects templates which it believes match the users input most, the user then selects the one they want. The template is then aligned with the input sketch using a camera calibration algorithm. From there the user traces the feature edges of the sketch on the computer screen, the user's 2D strokes are processed and interpreted in 3D to modify the edges of the template. The resulting template is shown and the user can refine initial surfaces using physically-based deformation techniques. Finally, new design edges can be added and manipulated through direct sketching over existing surfaces.

2.4.4 Model Animation

Sketch input has been used to animate models and a number of papers have been published looking at different ways of using sketch to animate models. Mao et al created a gestural interface for sketching out 3D animation for stick figures[29]. Their approach uses rapid 3D key framing and motion/timing control based on sketch input. Another approach was researched into using sketch-input for facial animation[33] which involves associating each stroke with a facial element, then the stroke is matched using templates, once it has been recognised, the differences between the template and the stroke are used to morph expressions on the model's face.

2.5 Gesture Recognition

Within the field of Sketch-based development, a common goal is to use sketch as a way of issuing commands and for the user to convey meaning in a particular application. Thereby moving away

from the traditional button and menu based interface which are commonly used today. A gesture based interface allows stroke input to specify commands.

Therefore, such a system would need to able to recognise gestures when they are drawn. Usually gesture recognition is template based, so a user can draw gesture 1 and it will match with template 1 in the system thereby causing that specific command to be executed. Templates can be represented in a number of ways such as simple bitmap representations, sequences of bits or anything which allows sketch input to be comparable. Thought should be placed into creating memorable and easy to draw gestures. Remembering the correct way to draw them still requires effort and practice on the user so the system should make it as easy as possible. An example of possible gestures is in figure 2.1.

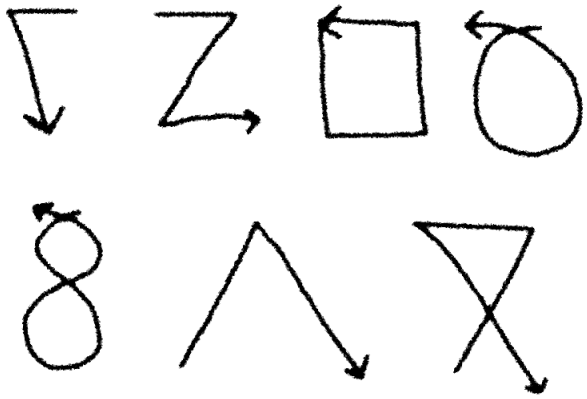


Figure 2.1: Example of gestures

There are a number of different ways of recognising gestures. Weesan Lee et al[25] propose the use of a graph based system where the symbols are represented internally as "attributed relational graphs that describe both the geometry and topology of the symbols." Then the recognition consists of finding the definition symbol whose attributed relational graph best matches that of the unknown symbol.

Rubine's[38] classical paper on his GRADMA system uses single stroke input only and its recognition system is based on the geometry properties of the stokes such as the initial angle of the gesture, the length and the angle of the bounding box diagonal, the distance between the first and the last point, the cosine and the sine of the angle between the first and last point and the total gesture length.

Hammond and Davis proposed the creation of a sketch recognition language which they named LADDER[16]. "The language consists of predefined shapes, constraints, editing behaviours, and display methods, as well as a syntax for specifying a domain description sketch grammar, ensuring that shapes and shape groups from many domains can be described." They then went on to define over 100 shapes in LADDER including UML class diagrams, finite state machine symbols, flow chart

symbols and various others. Avola et al[4] proposed another sketch based language based on XML, they also noted commonalities with LADDER and claimed that their system was able to label sketch strokes with a high degree of accuracy.

2.6 Miscellaneous

Sketch-based interfaces have found applications in numerous other areas not mentioned above. They have been used for road design[8][30] both using sketch-based input to define nodal roads and curve based roads respectively.

Zamora and Eyjlfsdttir[51] presented an application called CIRCUITBOARD. They propose a system which uses sketch input to draw digital logic circuits and tested out the resultant designs. Their system can recognise the full set of logic gates and provided a way of connecting them and defining the inputs and outputs to the design.

Anastacio et al presented a system for parametrising L-systems for the creation of plant structures[3]. User input sketches are employed as a way to define and manipulate global-to-local characteristics of L-system models. Eitz et al[11] used sketch input with regard to image deformations. Using user defined strokes to outline the region on the image for alteration, the user can then draw a new boundary stroke which will cause the image to deform matching the old boundary with the new one.

2.7 Summary

Sketch-based input is a very active and interdisciplinary research field. Many different applications have been researched for sketch strokes and while there are still many open problems left for solving, the interest in this area continues to grow. We can see this based on the growing number of papers being submitted to the Eurographics workshop every year[12][13].

3
Navigation

This chapter contains a brief overview of the subject of pathfinding, its background and its current state of the art. It describes overviews of the main algorithms used in current interactive entertainment applications.

3.1 Outline

Navigation is essential in many applications involving agents and traversable environments. It is used for searching for information in many types of graph structures and is useful in various areas of Artificial Intelligence. The type of graph structures can have an enormous difference to the quality of resulting paths generated and are mentioned in section 3.2. Pathfinding algorithms traverse the graph structures searching for a path to a target node. Pathfinding algorithms are discussed in section 3.3. Heuristic algorithms which estimate the cost of reaching a certain node are discussed in section 3.3.1.

3.2 World Representation

When it comes to navigation, how your data is represented in a 3D world is extremely important. Different representations have different benefits and pitfalls, and usually one is chosen over another

based on the application. Two considerations when a world or search space representation is being chosen is performance and memory usage. In interactive applications your final pathfinding solution will need to be as fast as possible and consume a reasonable amount of memory.

All representations are graph based, they all have nodes and connections called edges between certain nodes. Edges represent a possible transition between positions that can be taken. Some graphs can have a high amount of nodes, the larger the graph the more nodes that need to be stored in memory resulting in high memory usage. Also the larger the graph, the more search space that needs to be traversed to find paths between points and the slower pathfinding will be.

The goal of world representations is to keep the search space minimal but not too small so that it doesn't represent the world accurately. Different representations can be used for different aspects of an application, they just need to correspond to the same world. World representation can also have an effect on path quality, if the representation closely conform to the structure of the world, the quality of the resulting path will likely be higher than those representations which are distantly related.

The following mentions the most commonly used world representation data structures. Grid representation is the division of a world into rectangular tiles and are one of the most common representation structures. They can be other shapes such as squares, rectangles, hexagons or triangles. They are most useful in 2D environments and do not contain any 3D information. Should they be implemented in a 3D world, they will have to be modified. Within the grids themselves, there is a choice of using the edges, vertices or the tiles themselves for movement. If edges are chosen as their grid representation, the paths generated can suffer from quality issues with jagged results as they can only connect to cells vertically and horizontally. Tile and vertex based grids do not suffer from this. A negative of grids are that they do not scale well. A large number of grid cells are often required to represent large game worlds, thereby increasing memory usage and slowing down pathfinding. A positive aspect of grid structures is that they support random-access lookup[14]. A certain cell can be found in a structure in constant O(1) time.

A corner path representation, also called a polygonal map, creates nodes at the corners of obstacles. It is a non-grid based representation. If two points in a node are not blocked by interceding objects, an edge can be made between those nodes. The good thing about polygonal maps is that an edge in the graph can extend over a lengthy distance, making pathfinding much faster compared to a grid representation in which each cell would need to be traversed. A negative is that this representation can result in sub optimal paths with agents seeming to cling to objects as they traverse the map. They also take $O(n^2)$ time for generating edges between node pairs.

Waypoint graphs are another representation which are extremely popular in games today. They

are very similar to polygonal maps except that the node points can be placed in the middle of areas and do not have to be corners of objects. This solves the wall-clinging behaviour of polygonal maps. Unfortunately generating edges between the waypoint nodes takes as long as polygonal maps, at O(n^2) time. Also, while there are automatic methods for placing waypoints in a 3D world, usually a level designer will have to manually check and correct node placement. This can be a time consuming process. Waypoint graphics can suffer from problems with multiple agents passing by each other along the same edge between waypoints as edges can be quite long. The main benefit of them is their memory footprint which is lower than grid based maps.

Navigational Maps are the last world representation structures mentioned here and one of the more common maps in modern games today. Rather than map the obstacles like polygonal maps, they map the areas which are walkable using non-overlapping polygons. The result is that varying polygon sizes, usually all with the same number of sides e.g. 3 for triangular polygons, 4 for quads. An advantage of navigational meshes is that they support outdoor and indoor scenes equally well and allow you to reliably find the most optimal path between two points. Unfortunately they can also have very high memory usage, particularly in large and geometrically expansive environments[45].

3.3 Pathfinding

Pathfinding or path planning is the creation of paths through a suitable structure from point A to point B. They are a necessary part of any Game AI system for controlling agents in a virtual environment. Pathfinding algorithms search through a graph, starting at one point and traversing adjecent nodes until the final point is found or until the entire graph has been searched. There are many different pathfinding algorithms, some such as depth-first search will eventually find the final point if they are given enough time while others would be able to find it in less time. The following section details different pathfinding algorithms and their strengths and weaknesses.

Dijkstra's algorithm is a graph search algorithm which guarantee's finding the shortest path between two points. It works by calculating the distance between two nodes and then adding this value to the cumulative distance of the path so far. It repeats this process with all nodes around until it finds the final node or traverses the entire graph. When it finds the final node, it simply takes the path with the shortest distance and this is the shortest path. The performance of Dijkstra depends on the data structures used to hold the nodes and the size of the graph itself. The performance is $O(n^2)$ where n is the number of nodes in the graph. The worst case is when n = m where m is the number of connections from a node to its neighbours[32]. For use in real-time solutions, Dijkstra's algorithm is not used but it is important as it forms the basis for the A* algorithm.

The most commonly used pathfinding algorithm currently in use today is the A* search algorithm. Created by Hart et al[18] in 1968, pathfinding in interactive entertainment has become synonymous with A*. It is reasonably simple to implement, efficient and can be heavily optimised[32]. It is built upon Dijkstra and is identical except that it also takes into account heuristic values derived from the current node being tested and whatever heuristic algorithm being used. A heuristic is an algorithm which estimates the cost/distance to the final point. Using both the heuristic value and the distance cost travelled so far from Dijkstra, A* is able to find the shortest path to the final position and in much faster running time. The biggest factor in determining the performance of A* is the performance of its key data structures: the pathfinding list, the graph, and the heuristic[32].

When A* is being implemented in real-time environments, the algorithm is executed and a path for an agent is computed. The AI does not take into account any moving objects or dynamic environments so as the agent traverses the path, it checks to see if anything is in the way between the current node it is on and the next node. If there isn't then it proceeds to the next node, but if there is then it runs the A* algorithm again, taking into account the new obstacle. This strategy is known as local repair and it used extensively in games today[41].

A modification of the A* search algorithm was proposed by Anthony Stentz in 1993 which is called Dynamic A* or D*. D* is "capable of planning paths in unknown, known, and changing environments in an efficient, optimal, and complete manner"[43]. It is very similar to A* except in certain ways. For instance it starts at the final node and works at finding a path to the starting node. It changes the cost of edges between nodes as the program runs, if a node should become blocked by an obstacle then all it's neighbouring nodes have changes in cost reflecting this, raising their costs. This keeps the pathfinder away from the affected area if it can help it.

Many different variations of A* have appeared over the years, all dealing specifically with particular conditions they were developed for, algorithms such as Lifelong Planning A*[22], D*Lite[21] and Dynamic Fringe-Saving A*[44].

A relatively recent pathfinding method developed which is based on A* is Cooperative Pathfinding(CA*) [41]. There are number of differences between CA* and the original A* but the most fundamental was the idea that each agent would have access to another agents path. The paper introduced a data structure which it called the "reservation table". This structure organised all possible nodes in a particular graph, essentially a 2D structure but then added an extra dimension which represented time. In this way, when an agents path was being planned using whatever pathfinding algorithm it deemed necessary, most likely A*, it will then calculate at what time the agent would be at each node and check if that node is available or reserved by another agent. If it is free then it will reserve that node in the reservation table, otherwise a pause instruction would be issued until the

next step and this sequence would be repeated. The result of all of this is that agents move efficiently through the map, avoiding each other's paths and getting out of the way when necessary. The benefit of this is that while the initial performance overhead will be greater than executing regular A*, as the agents will not be bumping into each other, A* will not be run again.

3.3.1 Heuristics

"A heuristic is a rule of thumb: an approximate solution that might work in many situations, but is unlikely to work in all."[32]. Heuristics are used extensively in modern pathfinding, mostly because of A* based algorithm supremacy in creating reliable paths quickly. Heuristics are not only used in pathfinding, they are used in Evolutionary Algorithms and Neural Networks[23].

When it comes to heuristics in pathfinding, the higher the quality of the heuristic, the faster the path will be found. If a heuristic is perfect, it will return the exact distance to the target node and A* will go straight to that node. The runtime will be O(n) where n is the number of steps in that path. Most of the time, the heuristic will not be perfect, especially in complex search environments. It can either return an estimation that is low, in which case it is called an underestimating heuristic or high where it called an overestimating heuristic.

An overestimating heuristic could lead to A* returning paths which are not the best. As the heuristic returns a greater cost to the final node than it really is, A* will start paying less attention to the cost so far and more towards the heuristic as its result dilutes overall cost. The result is that while the algorithm may find the target node faster, it might not be the shortest path to be from the initial node to the final node.

An underestimating heuristic can lead to A* taking longer to compute a path but it will always return the shortest path. As the heuristic returns a lower cost to the final node than it really is, A* will start paying more attention towards the cost so far and less to the heuristic. This will cause it to start favouring nodes closer to starting node as they have a lower cost. Underestimating heuristics can cause A* to start behaving like Dijkstra.

The two standard heuristics for A* on grid like graphs are Euclidean distance and Manhattan. Euclidean simply gets the distance between two points regardless of data structure and uses this as the heuristic. Euclidean distance will always be underestimating and on complicated graphs can cause significant performance drops. Manhattan distance gets the number of nodes between the current node x-axis position and the final nodes x-axis position and adds it with the difference between the current nodes y-axis position and the final nodes y-axis position.

Part II

Project

4
Design

This chapter contains the design of the project, its goals and main components.

4.1 System Requirements

The following section lays out the requirements and goals of this project, broken down into their individual points.

- Take in and store user input from a sketch device in the form of strokes.
- Project points onto the surface of the terrain creating path.
- Parse the path correcting any continuity gaps.
- Check for path collisions with non-traversable terrain and correct.
- Allow agents to traverse user drawn path.
- Use the sketch pen device to move around a 3D environment.
- Allow agent selection with irregular sketch strokes
- Allow path editing, deletion and appending in real-time.

- Allow for multiple agent movement with agents path being planned relative to each other paths under the direction of sketch input.
- Allow sector omission from system pathfinder using sketch.
- Create debug viewer allowing additional information and geometry to be seen.

Throughout the project, effort was made to optimise implementations when time allowed, in cases where some implementations can be optimised further, I will mention possible improvements should anyone be interested in implementing any features.

4.2 System Design

This section describes the individual components of the project including how they interact and their main function. Figure 4.1 shows a diagram of the various components.

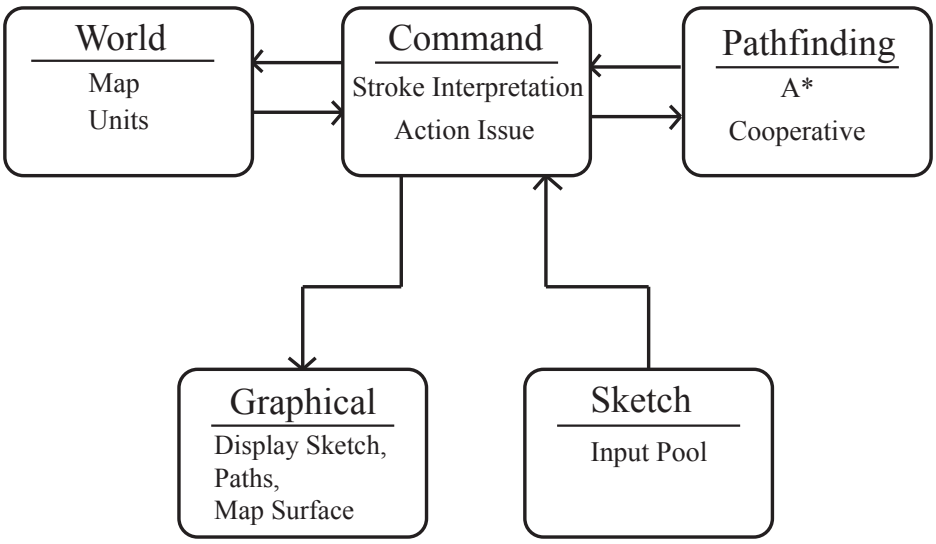


Figure 4.1: Diagram of main components of the system

4.2.1 Sketch Input

This part of the system is responsible for taking input from the sketch device, modifying it for use by the system and storing it when the action is complete. It is also responsible for transferring the points from the 2D screen space to 3D world space and recording the geometry the points interact with.

4.2.2 World

The World module is responsible for the world representation of the 3D environment. It also stores information related to the map and agents. For the world representation of the system, many data structures mentioned in the background section were considered. Finally a grid based map was chosen with vertices chosen as the points which would be represent cells. When agents are being rendered, they appear to be in the cell which the vertex represents. A grid map was chosen in the end as it would allow for random-access lookup of cells and this will be extremely beneficial when parsing paths which are mapped to particular nodes on the map. Grid based maps can have high memory usage for significantly large maps but as maps would only be less than 500 nodes in width and height, it seemed the memory usage would be negligible.

Polygonal maps were considered and while their memory usage is lower than grid based maps, their faults such as wall hugging and long look up times, especially as the map gets larger seemed too expensive for this system.

Navigational maps were also considered and while they are usually the primary choice for modern games today, like polygonal maps, the look up time made the deciding factor as it too grew larger as the size of the map increased.

4.2.3 Pathfinding

For the pathfinding module, the A* search algorithm was chosen. Its speed, reliability and its ubiquity in modern pathfinding solutions made it the perfect choice for the system. Its extensibility is also a positive allowing the system to take into account terrain heights and player orientation.

For group movement, rather than running the A* for each individual agent, it was decided that cooperative pathfinding(CA*) would be implemented which would allow each agent in a group to take each others planned path into account. The implementation of CA* differs in some ways from the CA* laid out in the original paper and take into account the sketch stroke drawn by the user.

4.2.4 Command

The Command module is responsible for storing the paths computed from the sketch module. It allows for the manipulation of the paths during runtime and provides functionality for searching and the attribution of paths to particular agents. It interacts with the World module and issues requests for information about agents and the map itself. It is responsible for interpreting the sketch strokes,

treating them for use as paths by agents and for the gesture commands which modify them. It sends updates to the graphical module for displaying the world.

4.2.5 Graphical Interface

The graphical interface module is responsible for giving the user visual feedback of actions in the scene. When the user uses the sketch stylus to draw or modify paths, or move around the environment, this module will show visual feedback of the actions performed.

5
Implementation

This chapter contains information on the implementation of all the various parts of this project that were mentioned in the past chapters. Specific tools used, with methods for implementing sketch input and the other parts of the project being explained.

5.1 Tools

For this project a number of different technologies were looked at.

- For programming languages, C++ was chosen as it is fast and is commonly used in interactive entertainment applications and graphics. It also has support from the standard template library(STL) which provided more ease of use.
- The sketch input device used is a Wacom Bamboo tablet[46].
- To gain access to data being read in from the tablet, WinTab drivers are used[15]. WinTab is a driver specifically made for the Microsoft Windows environment for communication between digitalising tablets and applications. A wrapper API for WinTab called bbTablet[6] was used as it provided a higher level view of the functionality of WinTab without the need to access its many low level functions.

- OpenGL is the graphics API used as it is widely supported and easily accessible using C++.
- Rather than have low level primitives representing agents and obstacles, GLEST was used. GLEST is an open-source real-time strategy game that is currently in development[9]. It was chosen as it is written in C++, uses OpenGL and allowed access to source code. It also has a level editor so it allowed the creation of levels to test different parts of the project.
- The development took place in Visual Studio 2008 on a Windows XP operating system.

5.2 Pathfinding

Throughout this project, the A* pathfinding algorithm is used. GLEST had its own implementation of A* for use in the game. It was necessary to rewrite this and extend it to account for a greater number of factors. As mentioned in the background section, A* works by combining both the total cost of getting to the current node from the starting node and a heuristic which will estimate the cost from the current position being evaluated to the target position. Cost is the distance it takes to get from node to node. When it is evaluating a position, it checks the distance cost it took to get to this node from the original position and it checks its neighbours to see if any have been searched before. If they have, it will then check if the current distance the search has traversed plus the extra cost of going to the neighbouring node. If this total distance is faster than the path which was taken to get to that neighbouring position when it was discovered last, then the new path has found a faster way to reach that position and it makes the current node the parent of the other node, thereby shortening the path. It continues this activity until it finds the target then iterates back along from final node to its parent and so on until it reaches the starting node and this is the shortest path.

The implementation of A* in GLEST did not have a number of features which were included in the end. The implementation did not take into account distance cost already traversed, it based its pathfinding ability on the heuristic solely. As such it was not able to guarantee the shortest path between two points but using the heuristic would eventually find a path. A* heuristics were extended taking into account terrain height, it was more prudent to stay on level terrain if possible then to ascend as the cost of transitioning between cells of a different height would increase costs. Euclidean distance was used as the heuristic as it would either be perfect or underestimating making it guarantee to finding the shortest path between two points and because agents can move diagonally, Manhattan distance is not used.

5.3 Sketch Input

For the sketch input, a Wacom Bamboo tablet was used. This tablet allowed the pen device to hover over a point without actually touching the pad and for the device to read this point. Wacom are able to do this using electromagnetic resonance technology. When the user starts touching the pad itself, the points begin to pool until the pen is lifted. The user can draw strokes on the pad and using the Wacom drivers, the system is able to read all the strokes pressed.

A method was written to check for events i.e. input information, where coming from the pad itself. If there is data detected, then it is pooled until the input stops. From this data, the position of the stylus on the pad(which is recorded as a 2D point) and the type of interaction with the pad is extracted. The type of interaction will tell whether the stylus touched the surface of the pad or simply hovered, if any of the buttons were pressed and if so which ones. When using the WinTab drivers, the input stream would stop mid-stroke and with how the pooling system was written, the points recorded up until the break would be designated as a stroke. For the stroke information to be continuously read from the device and for continuous input to be rightly designated as strokes, the checking system had to be modified. If two consecutive input checks are returned with no information being read from the pad, then all the points read before the breaks are considered part of the one stroke. Some inconsistencies also came from the amount of pressure being applied on the device itself. If the pen is pressed too lightly, the pad and subsequently the drivers would record no input whereas a simple hover of the pen over the pad records input well. This may be caused by faulty equipment or slight bugs in the drivers themselves when interacting with this particular type of sketch pad.

Once a series of points is designated a stroke, the data is quickly scanned for any consecutive duplicate coordinates and if any are encountered then they are removed. This is done for several reasons. As the points themselves will have to be projected into the 3D scene which is mentioned in section 5.4, the ray creation and intersection tests would be expensive especially when done on points that are duplicates. Also when a particular part of the map is selected with rays, duplicate cells will be removed anyway making the entire process wasteful. When the coordinates from the sketch device are being mapped to the screen, they are in the representation of 0 to 1 from the operating systems resolution and must be converted to the applications resolution.

5.4 Sketch Projection

When a series of points have been created for a stroke, they are in screen space and need to be converted to world space. A raycasting solution was implemented. Raycasting involves shooting rays

from the point in 3D world space. This is done by getting the current modelview matrix, projection matrix and viewport. Using these we can then reverse the graphics pipeline converting the points from screen space to world space.

A ray that projects itself into the 3D scene needs to be created. The following process occurs for each point in the stroke. A point in world space is created at exactly one unit away from the camera position into the scene using the reverse pipeline of the point. Using the camera position and the new point, a line of infinite length is constructed allowing the scene to be probed with the corresponding sketch points. Once all of the rays have been constructed from the original sketch stroke points, then tests need to be conducted to see which part of the map is colliding with the rays.

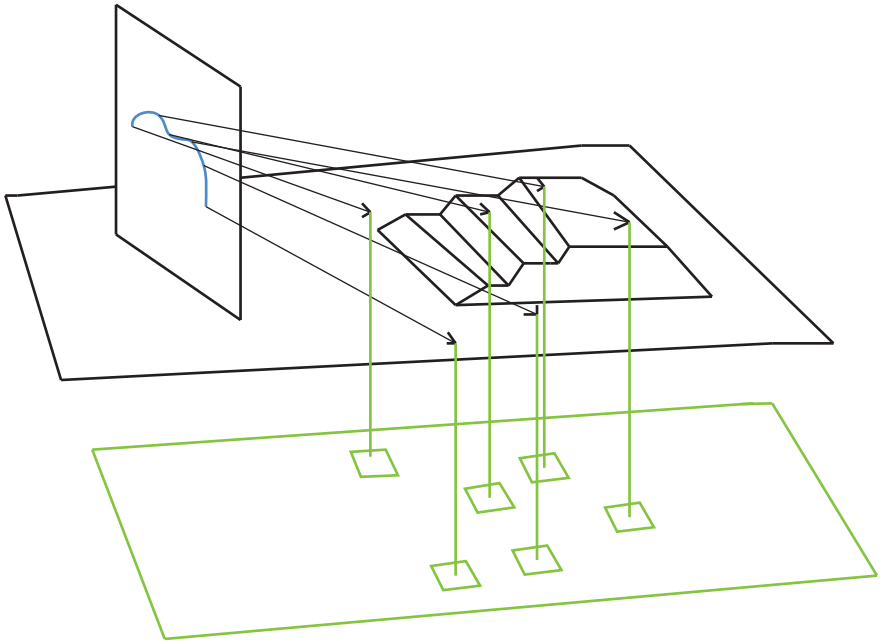


Figure 5.1: Sketch ray projection-the point of intersection is matched with a cell below

The terrain itself will be checked for collisions with the rays. As the terrain is made up of cells which form grids, the number of cells to be tested can be quite large. Rather than test for intersections against all cells in the map, a method was employed to reduce the number of tests that needed to be performed. When the environment is being loaded into the application, a note is made of the lowest height of the terrain. When the intersection tests are being run for each ray, a point of intersection is calculated when the rays cross the minimum height. Using this point, a bounding box can be created by getting the x and z coordinates of that intersection point and the x and z coordinates of the start of the ray being tested. Figure 5.2 shows a diagram of this process. As this bounding box encompasses

the only cells the ray could possibly intersect with, it eliminates large parts of the map from being checked.

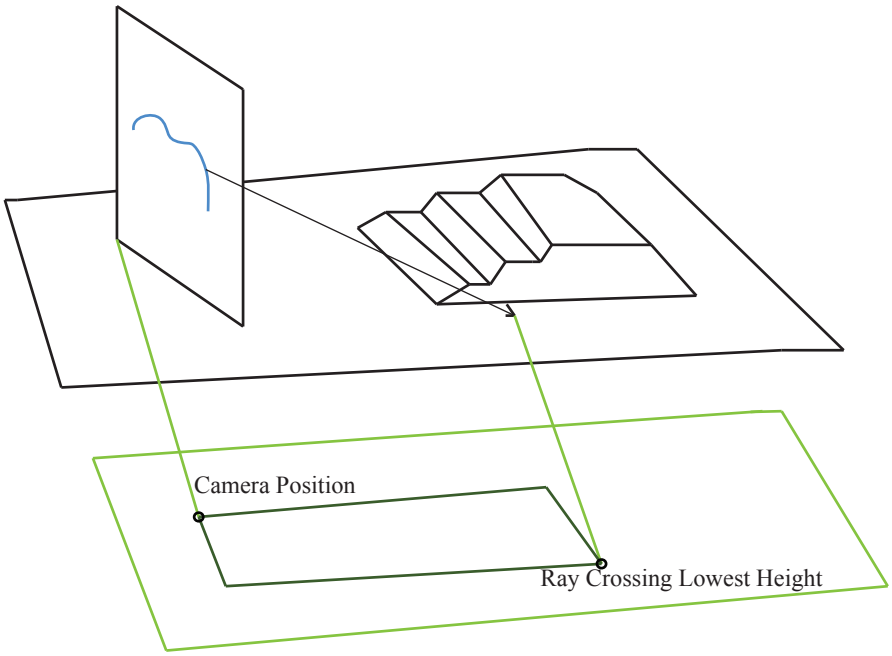


Figure 5.2: Bounding box created for ray

The system then iterates through the cells encompassed by the bounding box. For each cell, a polygon intersection test is performed. Using the points from the cell being checked, a plane which has the same orientation and position is constructed. A point of intersection between this plane and the incoming ray is then computed. From the cell's four vertices, two individual triangles are created. Then a triangle boundary test is conducted.

With the point of intersection, it must be checked to see if it is inside the particular cell being checked. To do this, tests are run with the two triangles which make up cells and original intersection point. The following method was implemented to test whether the point was inside a triangle geometrically.

There is a triangle made up of the vertices A,B and C and you have a point P which you want to test to see if it is inside the bounds of the triangle. The way it is implemented here is test each edge of the triangle and check if P is on the side which is inside the triangle. An example of such a triangle can be seen in figure 5.3.

If the edge BA is being tested against PA, the point inside the triangle we can test it against is C.

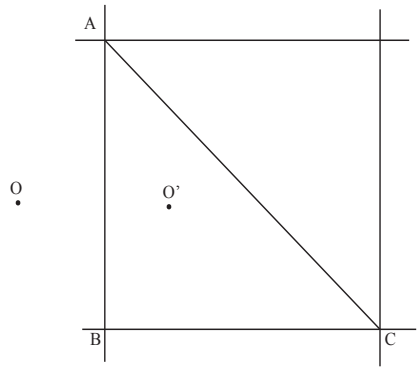


Figure 5.3: Diagram of the cell point

With the edge CB, the point inside the triangle is A and so on. This method checks a point to see if it is on the inside of all the edges and if it is then it is inside the triangle. If it fails any of these tests, the test exits.

$$D = BC \times PC$$

$$E = BC \times AC$$

$$result = D \cdot E$$

$$if \;\; result \;\; >= 0$$

$$then\ point\ inside\ triangle$$

If a cell is found to be colliding with a ray then that cell is noted. Once all the checks are complete for a ray, if multiple cells were found to have intersected with a single ray then the cell which is closest is added to a list containing the cells which are also colliding with other rays. As the stroke coordinates are stored in order of their input, the points at the start of the strokes are stored first and the points at the end of strokes last. When intersection tests are being conducted and the screen space points are being used, this order of first to last is used so the cell order also corresponds from first to last.

As the number of points being drawn on the screen are on the pixel level and can repeat, a number of the points in a stroke can hit the same cell at the same stage of the path. The result is that there are numerous duplicate positions in the path, all of which are adjacent to each other.

When this occurs, it is necessary to prune any duplicate positions from the list. Figure 5.4 shows

an example of the same cell being hit twice. This will insure that the agent is not stationary for periods of the path as the same position happens to feature a number of times and will also cut down on rendering costs. When the paths are being drawn, the same positions will be repeatedly drawn when there are numerous duplicates and the frame rate of the simulation can quickly drop.

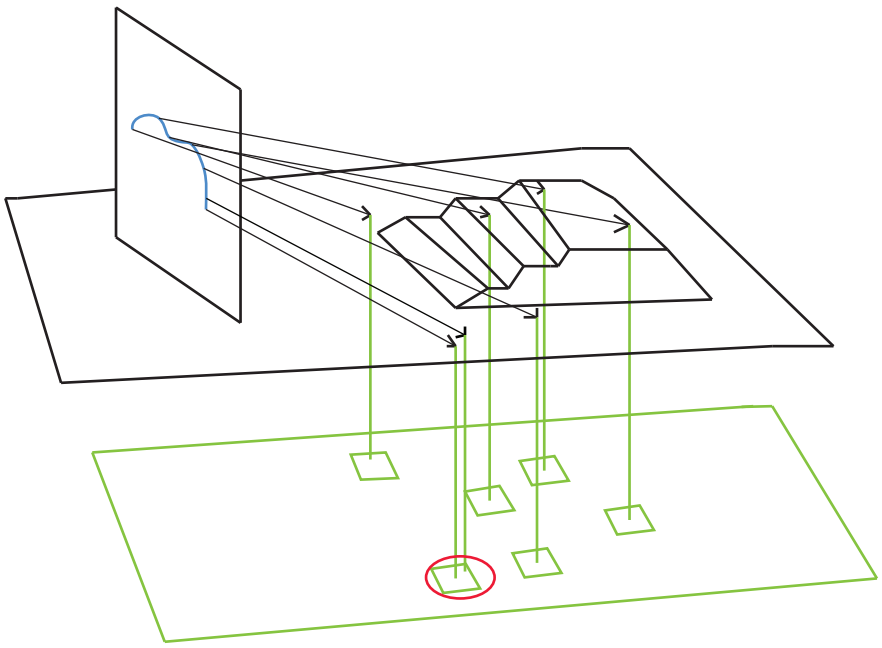


Figure 5.4: Ray Projection with multiple collisions in the same cell

It is worth noting that there is an opportunity for acceleration techniques with the raycasting. Typically when there are a high amount of rays being computed and intersection tests need to be conducted in scenes with a high amount of geometry, the drop in the frame rate can be substantial. In the system developed here, the number of rays were limited to the size of strokes drawn by the user and were not computed for every pixel in the screen like some other intensive raycasting or raytracing solutions. An acceleration technique was implemented whereby a bounding box covering the area of all possible ray intersections was created, allowing for a lot of intersection tests to be skipped. As the strokes are only computed once when they are drawn initially and do not continue in subsequent frames, it was not deemed a high necessity to implement advanced acceleration techniques as the potential benefits for performance would possibly be negligible or minute. This allowed time to be given to other areas. Spatial data structures such as Octrees, Binary Spacial Partition trees and bounded volume hierarchies may cut down on wasteful computation on parts of a scene not being selected. Construction of these structures are expensive however, and is often done as a pre-process[1].

5.5 Sketch Treatment

This section explains the process of treating the sketch path after it has been created. Quite a number of different methods need to check the path and correct it in many ways before it is read for use by an agent.

Figure 5.5 shows the stages of this process in order from the unaltered sketch path generated by the intersection tests to the final path being output by the path smoothing stage.

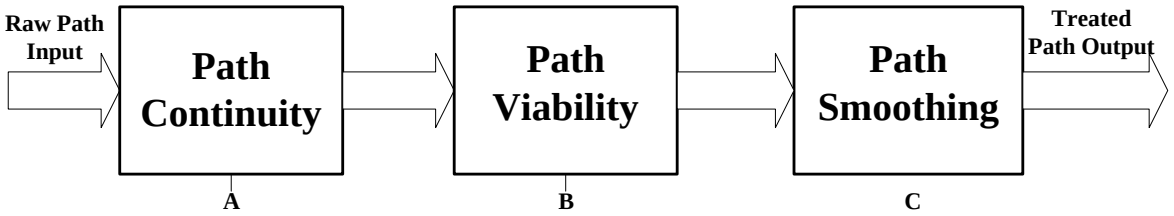


Figure 5.5: Diagram of the sketch treatment pipeline

5.5.1 Path Continuity

Now that there is a list of cells on the map corresponding to the sketch strokes, continuity gaps in the cells need to be checked.

The list is scanned with two pointers at consecutive slots in the list. As both pointers iterate along the list checking if the two cells they are currently on are adjoining in any way. If they are then the iterators continue checking until they reach the end of the list. If they are not adjoining then the gap needs to be closed. Gaps in paths will cause problems for agents as they iterate through the path themselves. When they come upon a gap, they will suddenly speed to the next cell in the path, possibly bounding over non-traversable cells in a very straight leap. There were two ways to go about closing the gap.

The first was to create a line between the cell at the start of the gap and the cell at the end of the gap and interpolate along the line checking the cells which the line passes through, adding them to the list as they appear. The second choice involved using A* itself to close the gap. The second option was chosen for implementation. The benefit of this is that usually the gaps are quite small and the pathfinding algorithm will be run for a very brief time, and A* guarantees that it will find a path if there is one. It should be noted that this version of the pathfinder will not take into account the orientation of the agents body or environmental factors such as terrain height. It is thought that

with these added conditions excluded, the resulting path found using the pathfinder would be close to what the original sketch stroke looked like. If the pathfinding algorithm did take terrain into account then you could get deviations from the original stroke where the pathfinder has found a quicker way to get from point A and point B. The point of using sketch as an input method is to give the user exact control over the paths agents traverse so the algorithms used should try to adhere to this goal.

Resampling of the sketch input when it was being read was not needed in this project as any inconsistent gaps between points would be closed by this path continuity process. It may be necessary for other applications that make use of sketch input to have their input resampled. Sometimes the spacing between points read in from the sketch device can vary and resampling could be used to reduce noise between points and to keep spacing between points consistent.

The result is that the path contains no gaps, with each cell being adjoined to its neighbours allowing for the agents continuous traversal over the terrain.

5.5.2 Path Viability

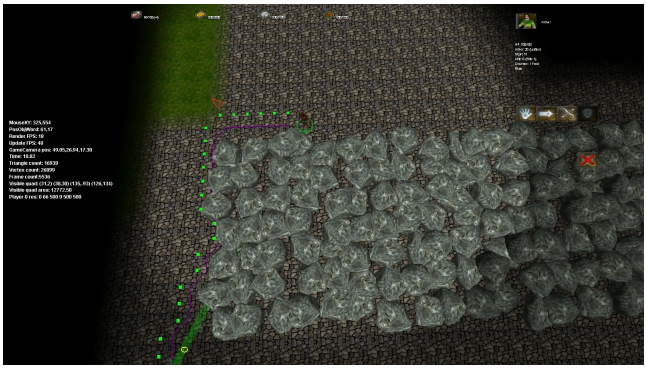
This section describes the part of the project dedicated to path viability, which is checking that a particular path is traversable by an agent. As sketch input gives the user the power to draw distinct paths over all types of terrain, certain paths drawn or possibly parts of the paths being drawn will pass over terrain that is not traversable by the agent selected.

It is up to the system to automatically correct these paths while trying to adhere to the original sketch drawn by the user if it is at all possible. In this system, each cell has a structure which says what type of obstacle(e.g. a tree, a river, rocks etc) or agent it has on it, if it has any at all. With this in mind, the system scans the now gapless path searching each cell in the path for cells which are occupied by obstacles or agents. If one is encountered then that cell is checked against the players width and length. As the system is using a vertex based grid map, that vertex can overlap with more than one cell, at a maximum four. So the the system must check the remaining cells to see if they are at all occupied by a agent or obstacles. As the cells dimensions are quite small, it is not necessary to check for collisions using bounding box type solutions but should the cells increase in size, some type of bounding box solution would be more than likely need to be implemented. As the world representation is already grid-based an axis-aligned bounding box would be appropriate for efficiency reasons.

When an obstacle is encountered then a pointer records the previous cell to its position. A second iterator starts scanning through the remaining list checking each cell. When it encounters a cell which is does not have an obstacle, it then records its position in the list. Now the first pointer points to

(a) Initial Stroke Drawn

(b) Unsmoothened Path



(c) Traversing the Path

Figure 5.6: Collision detection for path in the application

the last traversable cell before the non-traversable segment was encountered and the second pointer points to the first traversable cell after that segment.

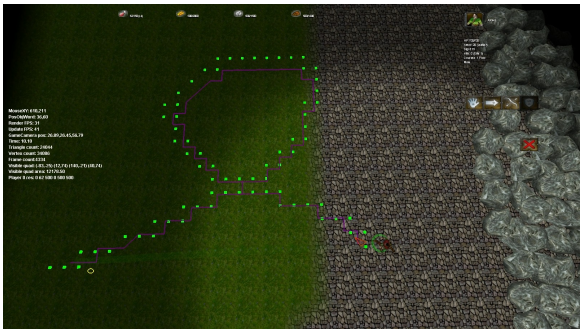
Using both points we now run the pathfinder and find a path connecting them both. The system now splices this new path into the original by removing the non-traversable and now bypassed cells, and inserting the new path to take its place. The agent can then traverse the new path and continue along the user designed sketch stroke. It is worth noting that multiple sections of one path can be non-traversable and all will be fixed by the system as it scans through the list. Figure 5.6 shows a sketch stroke being drawn and the non-traversable portion being corrected.

5.5.3 Path Smoothing

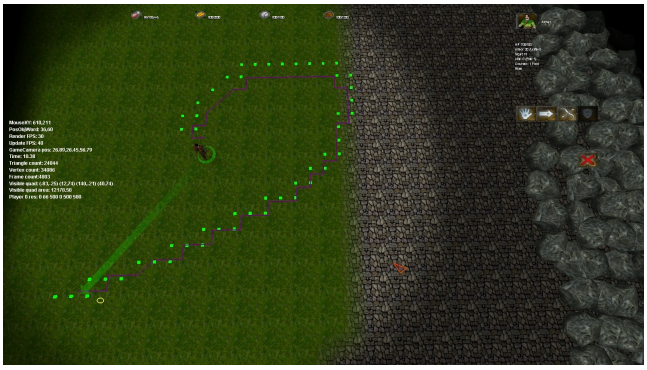
It became apparent that the paths which would be drawn using the sketch stylus were jagged in places and allowed for some extremely poor agent movement. As agents in the system are capable of diagonal movement, using the stylus sketch, there tends to be a lack of diagonal movement in places.

To combat this negative effect, an approach for path smoothing was implemented. Figure 5.7 shows examples of a non-smoothened path.

(a) Initial Stroke Drawn

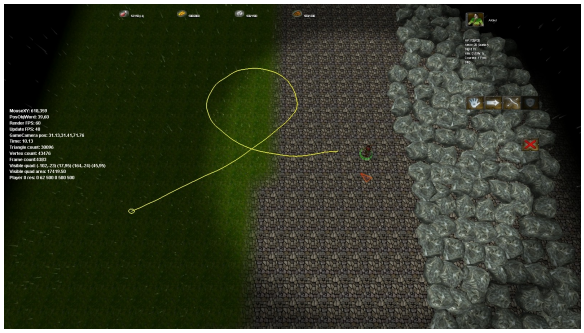


(b) Unsmoothened Path



(c) Traversing the Path

Figure 5.7: Path with smoothing off



(a) Initial Stroke Drawn

(b) Smoothened Path

Figure 5.8: Smoothing with lookahead of two

A pointer would start at the beginning of the path to be smoothened. A second pointer would

go a certain number slots ahead. How far the second pointer looks ahead is dictated by a variable which could be made available to the user to alter. For descriptive purposes, this variable will be referred to as the 'lookahead'. If the value set is greater than the remaining length of the path then the lookahead is set to the remaining path length. This lookahead is used throughout the smoothing process for setting the second pointer a certain number of slots ahead of the first pointer. Figure 5.8 shows the path smoothing function with a lookahead of two. A diagram of the two pointers iterating through an array can be see in figure 5.9.

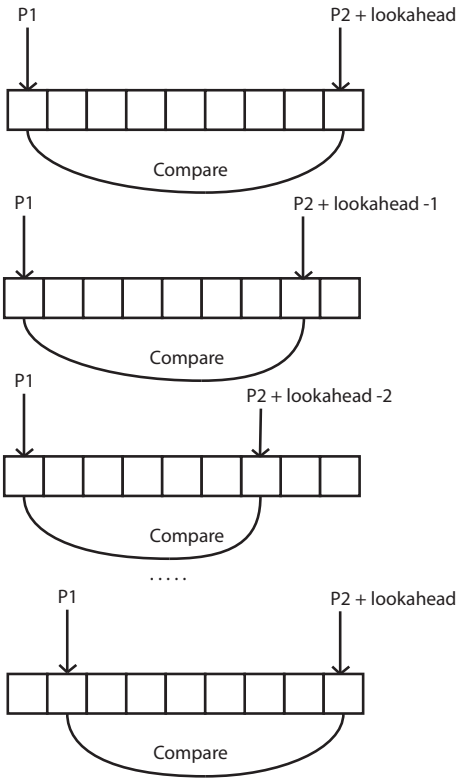
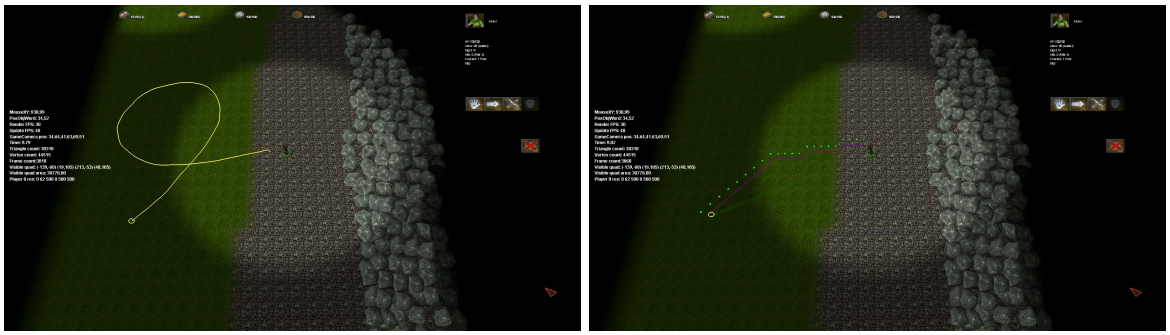


Figure 5.9: Pointer 1 and 2 iterating through path

The second pointer then starts iterating backwards, as it does this it checks each node to see if it is reachable from the node the first pointer is pointing to. If it is not then it continues to iterate back until it reaches the same node that the first pointer is on. From here the first pointer iterates to the next node. The second pointer again goes a certain number of nodes in advance as specified by the lookahead. It then starts iterating back again. If the second node does find a node which the first pointers node can reach, then all the nodes between the two pointers are removed from the path list.

It is by this process that any jagged nodes are removed from the list. Also if any loops are drawn by

the user and the lookahead length is long enough, the loop will be removed. The longer the lookahead the greater the chance that it could possibly be shortened. Full smoothing can be seen in figure 5.10. In the implementation of path smoothing in this project, the lookahead value was set to one as this would remove any jagged movements that the agent would make and allow the user to draw any path he/she wishes. Also, The longer the lookahead value, the greater the computation time for path smoothing.



(a) Initial Stroke Drawn (b) Smoothened Path with Large Lookahead

Figure 5.10: Smoothing paths in the application

Another possible alternative to path smoothing would be to map the path to a bezier curve and then plot this curve to the grid. Various path smoothing techniques have tried using bezier curves in pathfinding stage itself[50] and for autonomous vehicles for smooth steering[48]. As this system tries to adhere to the path drawn specifically by the user, bézier curves were not appropriate.

5.6 Sketch Analysis

This section contains information on certain commands which can be issued using the sketch stylus and how they work. After the treatment process has concluded, the system will now examine the sketch stroke and determine which command the user was trying to initiate, if there was one at all. This section explains the gesture recognition system implemented. As sketch is used in a certain number of determined ways, the system will now try and narrow down which command the user is trying to invoke. It does this in a number of ways. Firstly, as sketch is primary used to control agents in the environment, the system checks whether any agents are currently selected. If there are some then it assumes that the user was trying to issue a command towards the agents selected. The current state of the agents is also important. If they are stationary, then only certain commands are possible just as there are only certain commands if the agents are moving. A flow chart of the gesture

recognition stage can be seen in figure 5.11.

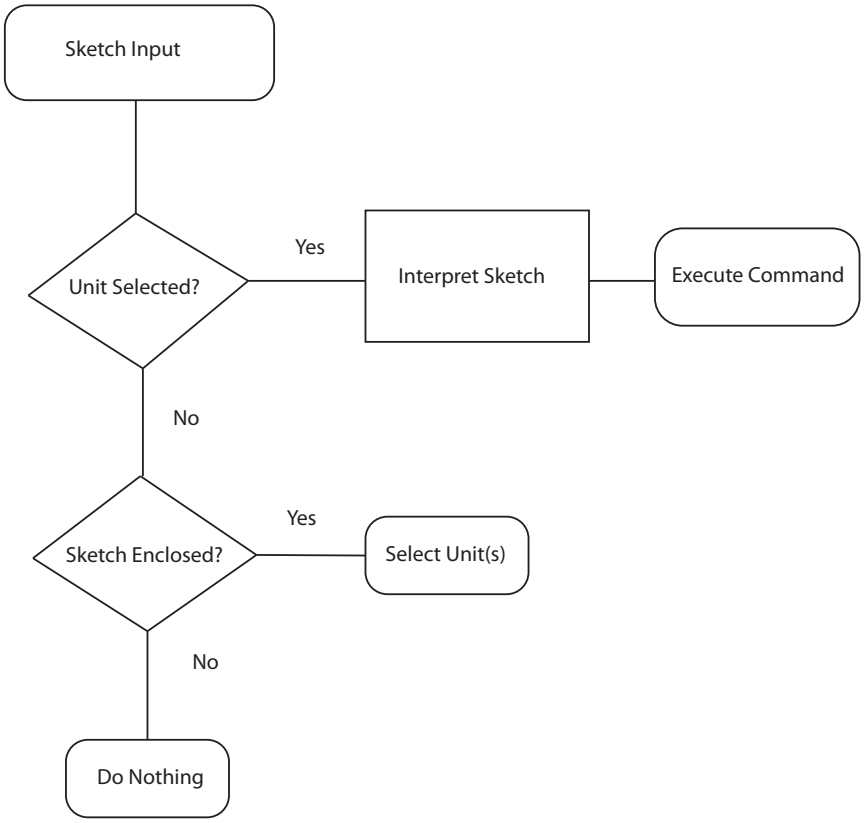


Figure 5.11: Flow chart of the gesture recognition stage

In the following subsections, each possible command and action will be discussed and the conditions which are to be met for them to be initiated. For many of the sketch commands to work the system needs to recognise the stroke patterns. As noted in the background, there are many different ways to do this and pattern recognition is still considered to be a problem with active research continuing in the area constantly.

Template matching was considered but as irregular shapes would have to be supported, using pre-determined gestures would be limiting in the way sketch strokes could be used. In the end, a different approach was implemented than the methods mentioned. As the commands issued using sketch would be mainly focused on the paths themselves, gestures were developed which were thought to be memorable and intuitive so the user would be able to use them without much effort.

Gestures would be recognised in the form of intersection tests between strokes and the context in which the strokes are drawn. When the user draws a sketch and after it has passed through the

treatment process, each cell of the stroke would be broken into continuing line segments. Figure 5.12 shows an example of this.

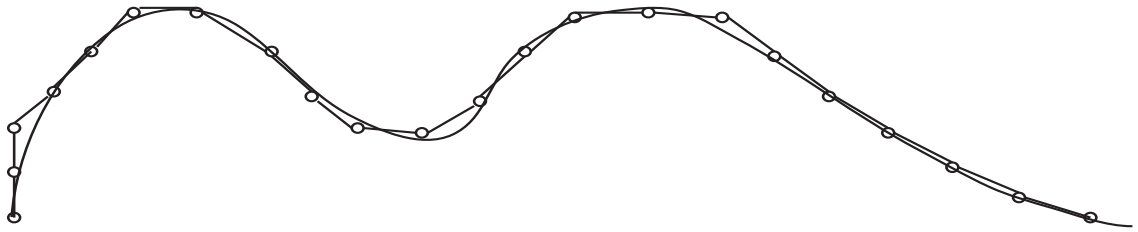


Figure 5.12: Path broken into segments

The paths of agents currently selected will also have their own line segments corresponding to their paths. When a stroke is drawn a number of tests are run. Line segment intersection tests would be executed, iterating through the stroke just drawn and checking each segment for intersections with the path of the agents currently selected. These tests are conducted looking for specific points of intersection if there are any. The points that comprise the line segments are 2D, representing the x and z axis. Two intersection points can be seen in figure 5.15.

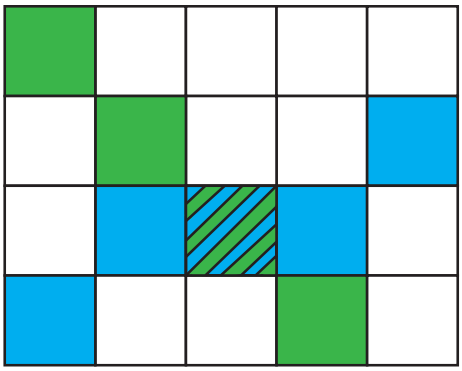
Line segment intersection tests are used because they give a certain reliability that other methods do not and were chosen over curved solutions such as bézier curves or clothoids as the intersection tests between line segments are less computationally expensive. For example, as the world representation in use is grid based, you could possibly attain both strokes with reference to grid cells and check for commonalities. This would be more efficient than running line segment intersection tests as the paths being checked would only have to look for common cells in both. There is a problem here in that while such tests may work for some strokes, they will not for others. In Figure 5.13a you can see two paths intersecting at a common point being detected, but in figure 5.13b we can see that despite the two lines intersecting, when mapped to the grid they fail they intersect.

The following subsections will detail how intersection tests are relevant to their commands.

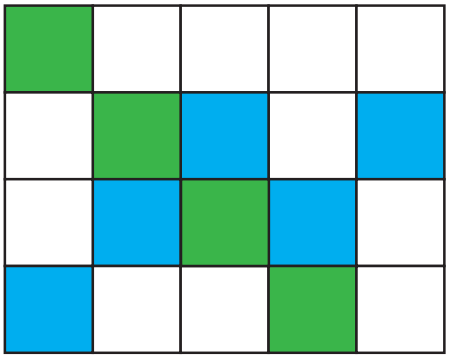
5.6.1 Selection

For the selection of agents to work using sketch gestures, there needs to be no agents selected. If there are agents selected when a selection like gesture is drawn, a different action could possibly be executed as the context for selection is not correct.

For selection to be executed, the user must draw a sketch stroke which intersects with itself. This



(a) Two lines intersecting at the same cell



(b) Two lines missing the intersection as the lines do not share any cells

Figure 5.13: Grid representation of intersection

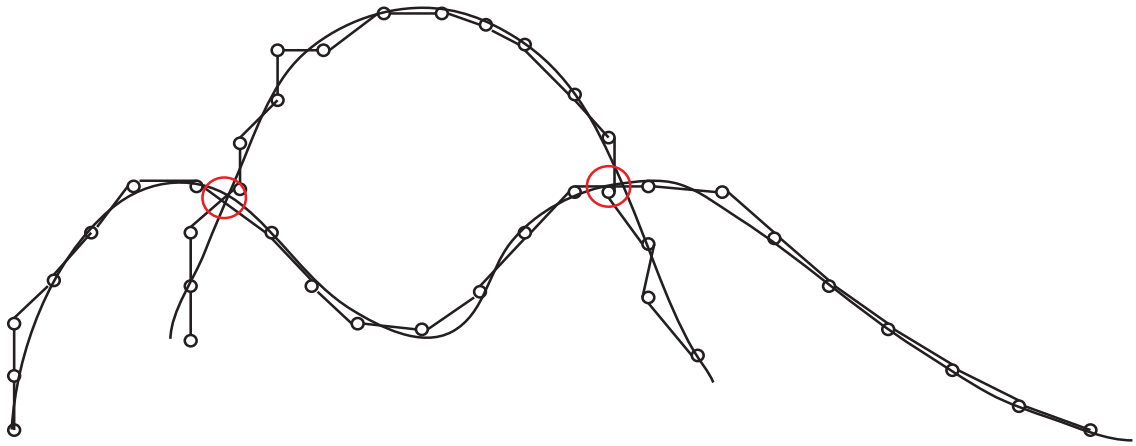


Figure 5.14: Two lines being checked for intersections - Intersections detected at red circles

means that the user has to draw a loop. The sketch stroke will be checked for self-intersections and if there is one then one of two things can happen. If the user has drawn a loop around an agent then the agent will be selected, if it is not encapsulating an agent then the section of the map selected will be become an omitted sector. This will be explained in the Sector Omission section later in this chapter.

The selection process works in the following way. When a command has been confirmed as a selection command, a bounding box is immediately constructed, encapsulating the self-intersecting loop. Then the cells inside in the bounding box are accessed, if the cell contains an agent then it is

checked to see if they are inside the self-enclosed loop. It is checked by iterating from the cells position along and negative and positive x and z axis checking to see if it is surrounded by a sketch mark. If it is and there is an agent on the cell then that agent is added to a currently selected list. This method is effective put can lead to some false positives if the sketch stroke is sufficiently complex where it encapsulates agents but not inside the sketch stroke. Except in these rare cases, it works exactly as expected.

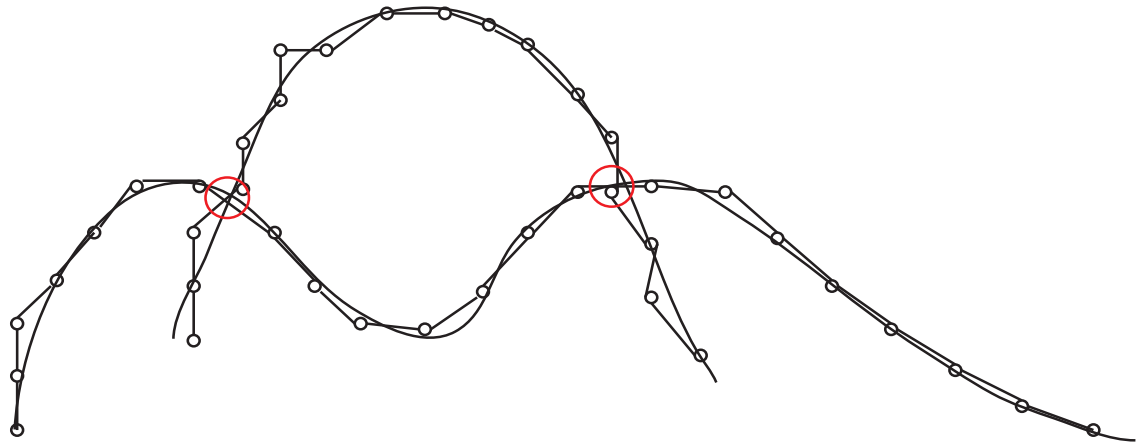
5.6.2 Editing

For the editing of an agents paths, the agent in question should be selected. The method used to do this is based on oversketching the path being edited. As shown in figure 5.17b, the sketch is drawn by the user, one which intersects the current path in two places. That sketch is broken up into line segments and it and the path selected are checked for intersections. Figure 5.15a shows two paths being intersected at two points. The red circles show the points of intersection.

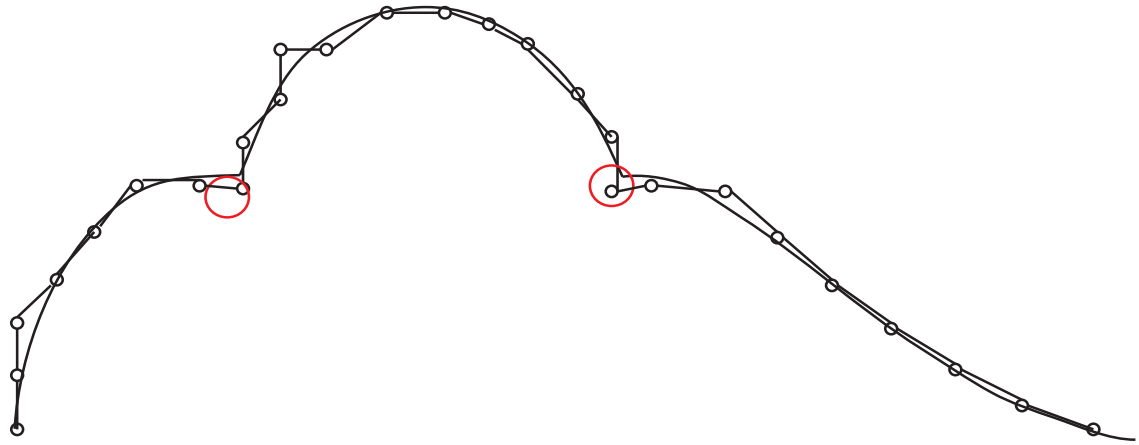
When the two points on both lines are found, both paths need to be spliced and swapped. On the path being edited, all the information between its two intersection points need to be removed. Then all the path information between the two points on the corresponding line which is doing the editing need to by inserted into the original line. Figure 5.15b shows the result of this. The resulting path will be the original path with a new section created using sketch and in real-time.

It is important to note that the order in which the editing stroke is drawn is important. For instance, if the agent is traversing along a path from A to B and you draw an editing stroke that intersects on the B side and intersects again on the A side, what you will have is two paths which are going in opposite directions. Figure 5.16 shows an example of this.

The problem with this is that when the two paths are being spliced into each other, it can introduce discontinuities into the resulting path and irregular agent movement will result. If the order of the positions in the new path is going in an opposite direction then the agent will jump to the end of the new path splice, start traversing back along the new addition to the line only when it gets to the end of this new addition, it will again jump to the section of the original line after the splicing took place. So when the two paths are being merged, the order of the new addition needs to be accounted for and reversed as it is necessary to avoid any contaminated paths. Figure 5.17 shows the resulting editing operation in the application.



(a) Two lines being checked for intersections - Intersections detected at red circles



(b) Path after editing command - original path now modified

Figure 5.15: Line Segments for Editing

5.6.3 Deletion

Deletion functionality was added giving the user the power to delete sections of the path while an agent is traversing it. To do this, a new gesture was created allowing the user to both select the section for deletion and to make it distinct enough for the system to detect that the delete action was being requested and not the editing command. For this command, a self-enclosed loop was used to signify deletion. The agent, whose path was to be deleted would be selected and the user would then draw a loop which intersected the agents path at two points. Essentially this is selecting the area of the path for deletion using the sketch device and making sure the selection sketch is self-enclosed.

When the sketch is drawn by the user, after it has been broken into its corresponding line segments

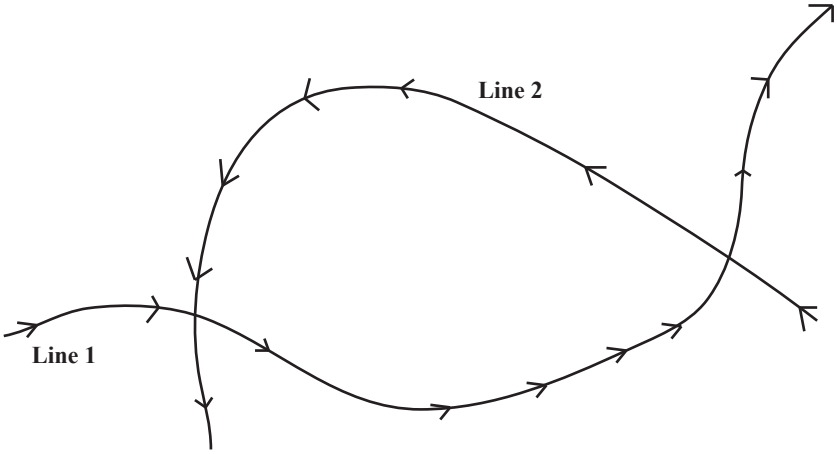
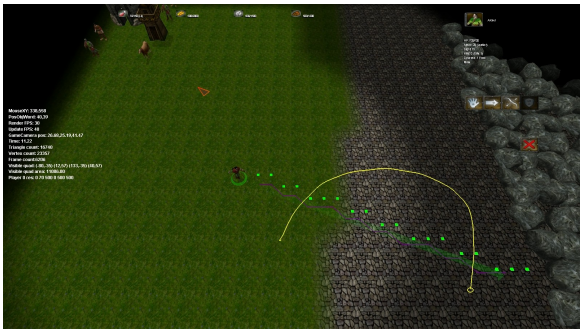


Figure 5.16: Two lines with their constituent points going in opposite directions

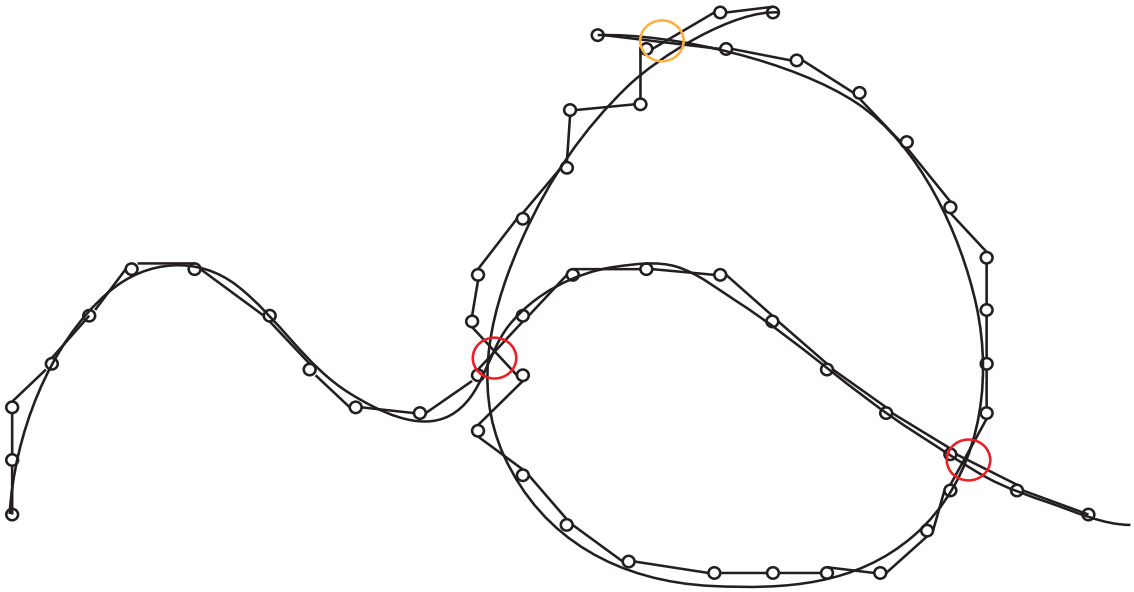
(a) Initial Path



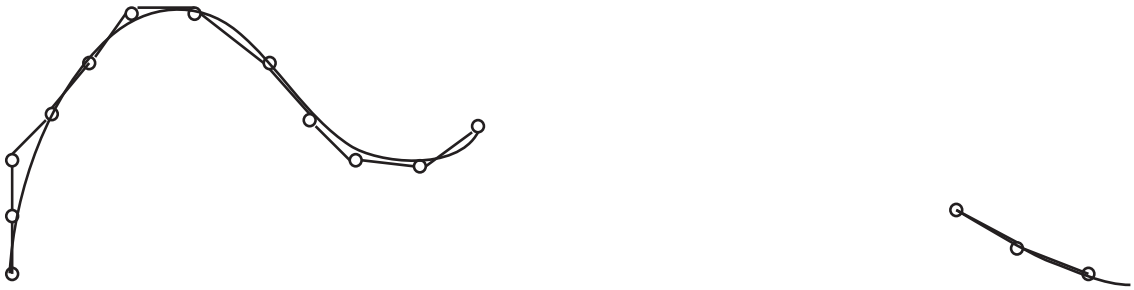
(b) Drawing Editing Stroke

(c) Result of the Editing

Figure 5.17: Editing path in application



(a) Deletion gesture - Two lines being checked for intersections - Intersections detected at red circles

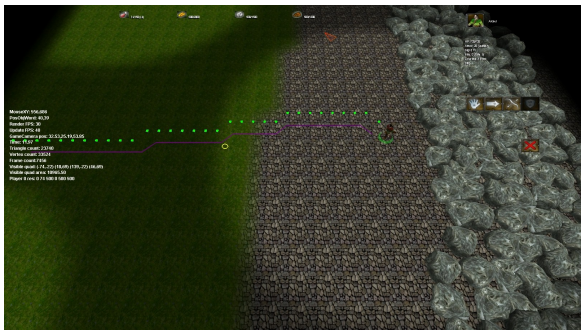


(b) Path after deletion command command - End of path now before gap

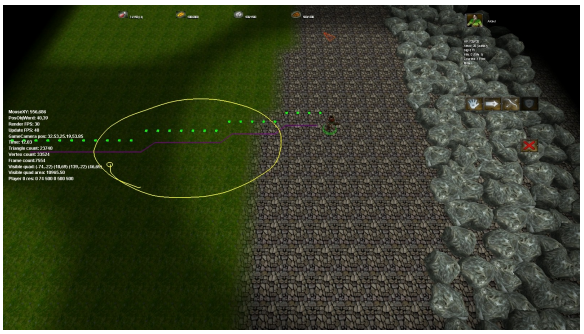
Figure 5.18: Line Segments for Deleting

and intersection tests have been run between it and the agents path, like the selection method it is checked to see if it self-intersects at all. If this is the case and if two points of intersection were found then the positions between those two points are erased. Figure 5.18a shows two lines being checked for intersection points, with one line being self-enclosed. From figure 5.18b, it can now be seen that the portion of the path which was encapsulated by the enclosed line has been removed.

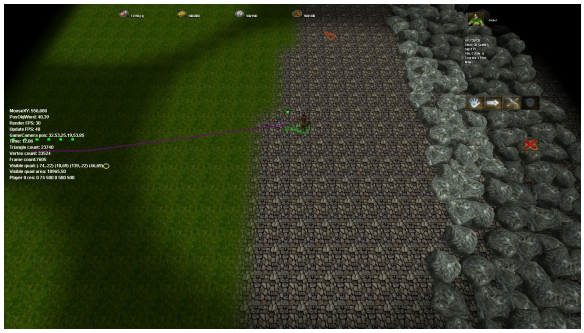
The agents target point is then updated to the last point in the current path before deletion. This is to prevent the agent from jumping between points over a long distance. The user can still add a new path to the place where delete section is and the agents movement till continue along the new path. The delete opertion can be seen in figure 5.19.



(a) Initial Path



(b) Draw Delete Gesture Around Section of Path



(c) Section of Path Deleted

(d) Agent Halt at Empty Section

Figure 5.19: Deletion in the application

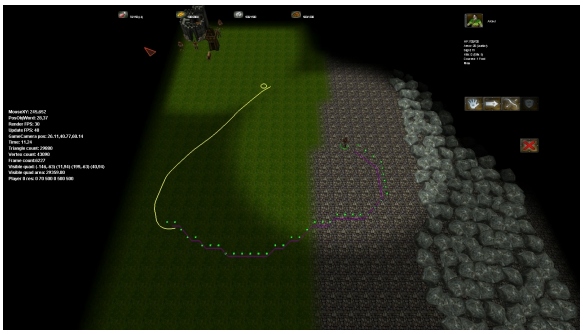
5.6.4 Appending

For appending a new sketch-drawn path to a path already being traversed, a certain number of conditions must be met. Like all of the possible commands for modifying paths so far, the agent whose path is to be modified needs to selected. Once it is, then the user can append a new path segment to the current. When a new sketch line is drawn, if none of the other checks for the sketch commands are met then the location of the start of the input is checked. An intersection test begins and if a point of intersection is found and it happens to be in the last two segments of the current path then the new sketch line is appended to the end of the agents path, extending it by however much the user drew. Figure 5.20 shows the append operation.

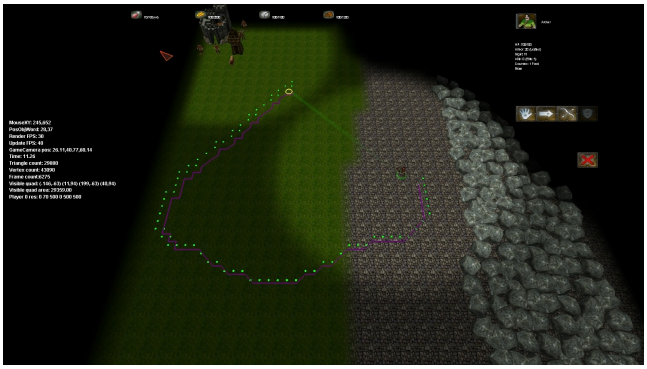
5.7 Cooperative Pathfinding

This section contains information on group movement using sketch as implemented in this project. Rather than use local repair A* which was briefly described in the background section(section 3.3), a

(a) Drawing Initial Stoke



(b) Drawing Appending Stroke



(c) Result of the Appending

Figure 5.20: Appending in the application

different approach was implemented. As local repair A* contains now functionality for avoiding other agents, cooperative pathfinding was implemented.

The implementation of cooperative pathfinding in this project is not the same as demonstrated by Silver[41]. Silvers implementation uses a heuristic for the pathfinding. The heuristic used is the A* algorithm itself and is termed “true distance”. What he means is that A* will find a path between points A and B and it will be the shortest if used properly. In this way, if you get the length of this path, it is a perfect heuristic as it will always return the exact distance, never under or over-estimating. This part of the algorithm was not implemented in this project as running A* as a heuristic would carry performance penalties. What was implemented is the “reservation table” as used by Silver. This table in terms of a grid based map would be a full copy of the grid map itself, a 2D structure except there would be multiple copies of these maps all according to a specific time. The reservation table is essentially a 3D data structure with multiple copies of the 2D map according to certain times. Figure 5.21 shows a diagram of the reservation table.

The idea is that at a certain clock cycle, the system will know which cells will be available and

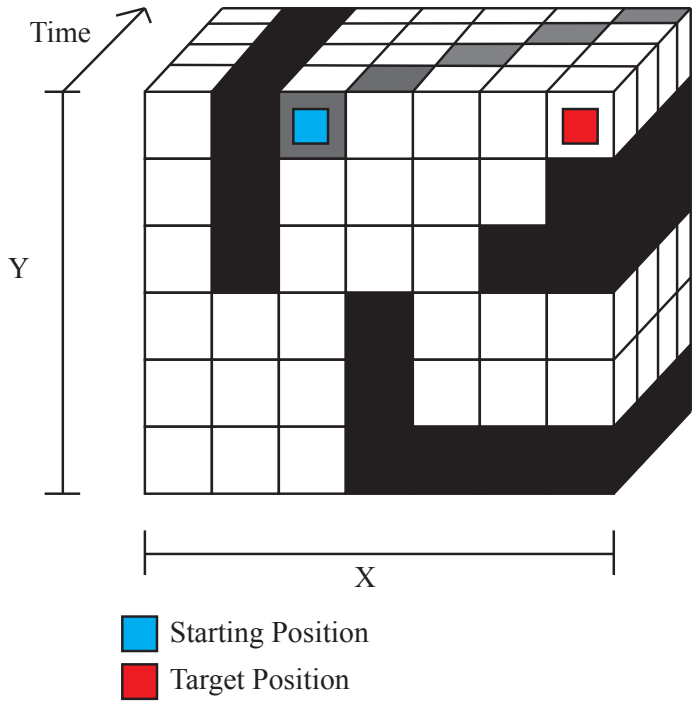


Figure 5.21: Diagram of the reservation table

occupied and in this way it is possible to plan paths that will be empty at particular times, thereby avoiding collision between agents moving across the map. Using the selection command described in section 5.6.1 multiple agents can be selected. Then the user will draw a path using the sketch stylus. For cooperative pathfinding to work, an agent must compute their path first. With this implementation, the user nominates a leader and that agent will be the first to have its path allocated. This occurs by finding which agent is closest to the path just being drawn, so which agent was that path intended for. This is done by getting the distance between the start of the sketch path and the position of the agent. Once an agent has been chosen then a counter is enabled and the reservation table is created. The system iterates through the paths and every position is added to the table but in different versions of the map, each selected by the particular time it will be there at. New time versions of the map are created as they are needed. Once this is done, the rest are to have their paths computed and added to the reservation table. For the rest of the characters I simply used A* to find a path to the final position as indicated by the sketch stroke.

They could be forced to adhere to the sketch stroke as drawn by the user, but for this implementation it has not been included. Rather, this implementation was testing the capabilities of a reservation table based solution and using sketch to test out its functionality. When the agents paths are being

added to the reservation table, they check if position is free on the map when they will be there. If it is not then the agents will add a pause for one iteration and see if it is in the next iteration, when it is free then they continue iterating through the path and adding it to the table.

The point of this is while the start can be quite intensive(especially if A* is used as a heuristic) it stops the pathfinding algorithms from running again as they will not collide with any static obstacles or any agents. In static environments, the initial computational overhead could be lower than a group of agents running A* every time their paths cross.

5.8 Sector Omission

This section contains information on the implementation of sector omission. Sector omission is a part of the map which is omitted from future executions of the systems pathfinder or of the users sketched paths. The concept behind this is that it being possible for the user to customise the way in which the system's pathfinding module runs using sketch strokes.

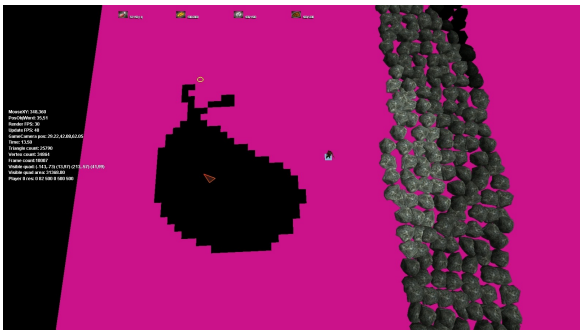
A user customising certain parameters of the pathfinding module while the system is in runtime and using sketch has very powerful implications. It could open up interactive applications to tailoring their functionality to the users criteria, giving them more control over agents.

When no agent is selected, if an enclosed loop is drawn, the cells inside are considered omitted. For this implementation, to allocate certain cells as non-traversable, as the implementation of maps within this system allows for status of cells to be stated, an added parameter records whether it has been designed for omission from subsequent pathfinding tests. This also allows for the action to be reversed easily once a gesture is created that issues that command and contains the area or re-admission to the pathfinding module. Figure 5.22 shows sector omission in the final application.

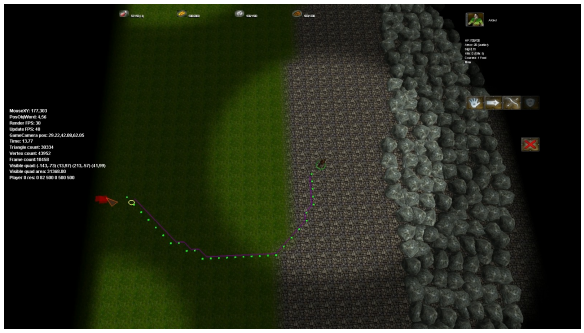
5.9 Rendering

For this project everything is rendered using OpenGL. A debug mode was created for showing terrain of different heights in different colours so it would be easier to see if certain parts of the system work correctly. Occupied cells are coloured differently, cells which are in agents paths are displayed and both can be updated as the system progresses. Omitted sectors are also viewable in debug mode letting us see parts of the map which have been set by the user as being omitted by the pathfinder. As the map's cells are being iterated through and their appropriate status are being rendered, data structures containing currently selected agents and their paths and omitted sector cells have to be

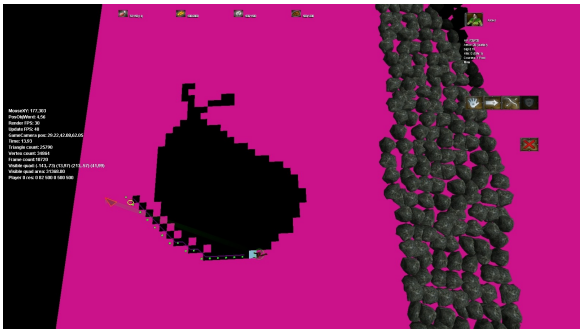
(a) Initial Stroke Selecting Sector



(b) Debug Mode - Showing Omitted Sector



(c) Pathfinder avoiding Sector



(d) Debug Mode - Path Avoiding Sector

Figure 5.22: Sector omission in the application

checked, these values were surrounded in bounding boxes to limit these structures traversals and performance noticeably improved.

For non-debug mode, certain graphical parts were written. Points which are being drawn using the sketch device and being pooled are drawn to the screen regardless of whether the stroke is completed. Once the stroke has been submitted to the command module for interpretation, this screen representation of the stroke is cleared. Paths were displayed using diamond shaped indicators and the line segments that made with paths is displayed using simple lines. Also, when the input is being read in from the stylus, the current position of the pen relative to the screen is displayed using a cursor.

5.10 Miscellaneous

This section contains information about other functionality implemented which should be noted.

When a sketch stroke has been drawn, it is very likely that it is not exactly where the agent is positioned. To make sure the sketch is still possible to reach, any gap between the start of the sketch

stroke and the user must be closed so the pathfinding algorithm is run.

To make sketch-based input a viable way of controlling the movement of agents, it is necessary to find a replacement to using the mouse for moving around the 3D environment. To achieve this, controls for the systems camera were created specifically for the sketch stylus. The Wacom Bamboo tablet has two buttons on the pen stylus itself allowing more information to be received from the user. A description and diagram of the controls are in Appendix A.

6
Evaluation

This chapter contains the evaluation of the project. It lists the performance at stages when sketch is being used in terms of the number of frames per second(FPS). The computer system being used to benchmark the project is using a Quad Core Intel Xeon CPU with each core clocked at 2.67GHZ and 3 gigabytes of RAM. The GPU is a NVIDIA Quadro FX580 with 512mb of memory. It is running Windows XP Professional. The application FRAPS[7] was used to calculating the number of frames per second as various parts of this project are being run. An application based frame-counter in GLEST itself was used to back up FRAPS results.

6.1 Performance

As this system is supposed to be real-time, the frame rates will be able to show how applicable sketch based path control is to real-time applications today. 30 FPS is considered the frame rate to be aimed for in modern computer games today.

Here the frame rate is checked when the sketch input is being read in from the device and the data is being pooled. The points are being used to create rays which are being projected into the scene and the resulting intersections are being stored. The strokes are also being drawn to the screen as they are being read. Figure 6.1 shows a chart mapping the frame rates against the seconds as they increase. As the counter begins, the application is idle with no agents in motion. The frame rate here

is 321. From seconds 1 - 4 the selection of a agent occurs and from seconds 5 - 15 the stylus is drawing a stroke on the screen. The minimum value recorded during this time is 315 FPS(frames per second), well above frame rates needed for real-time applications today.

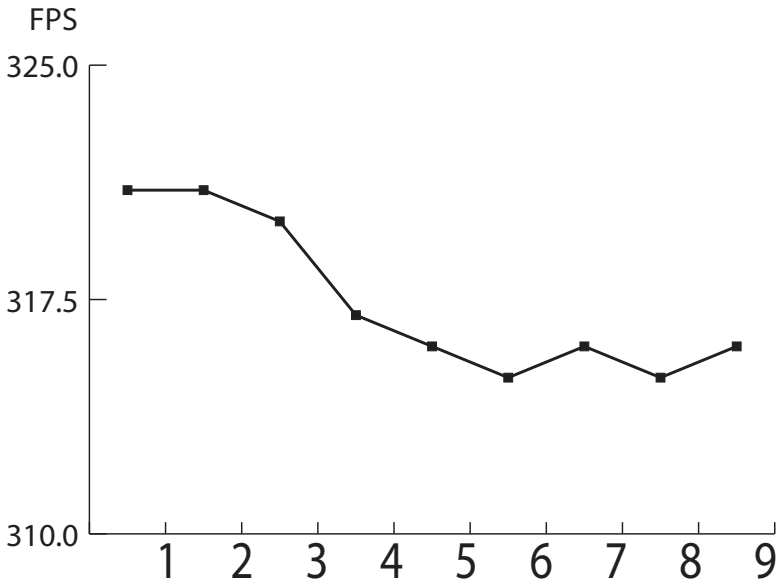


Figure 6.1: FPS graph as sketch input is being read

The sketch treatment process needs to be benchmarked now. As it does not involve reading input from the sketch device, it will be much faster than the sketch input phase. Figure 6.2 shows the frame rate during the sketch treatment process, the self-intersection tests and the agents traversing the paths after they have been added. The Second 0 at the beginning of the graph is a continuation of sketch-input being read in pooled as mentioned in the last chart. Seconds 0-2.5 is the section of the chart where the path is being treated and checked for self-intersection tests. The frame-rate starts increasing slowly at 3.5 seconds to 313 FPS and 312 FPS as the agent begins to traverse the path and the frame rate stabilises in this range. Again this frame rate is above the minimum for real-time applications.

Here the sketch-based editing is benchmarked. It is the most intensive sketch modification command as it requires splicing the original path and the new stroke into a resulting path for the agent. The other two processes mentioned so far are also included. These are recieving the sketch input data from the device, the raycasting of this data into the scene, the treatment process with the resulting path and the intersection tests between the agents path and the sketch path. Figure 6.3 shows the frame rate mapped against the seconds for the editing operation. The sketch strokes start to be read in from the device start at 0 seconds and the process just described begins. During this period the

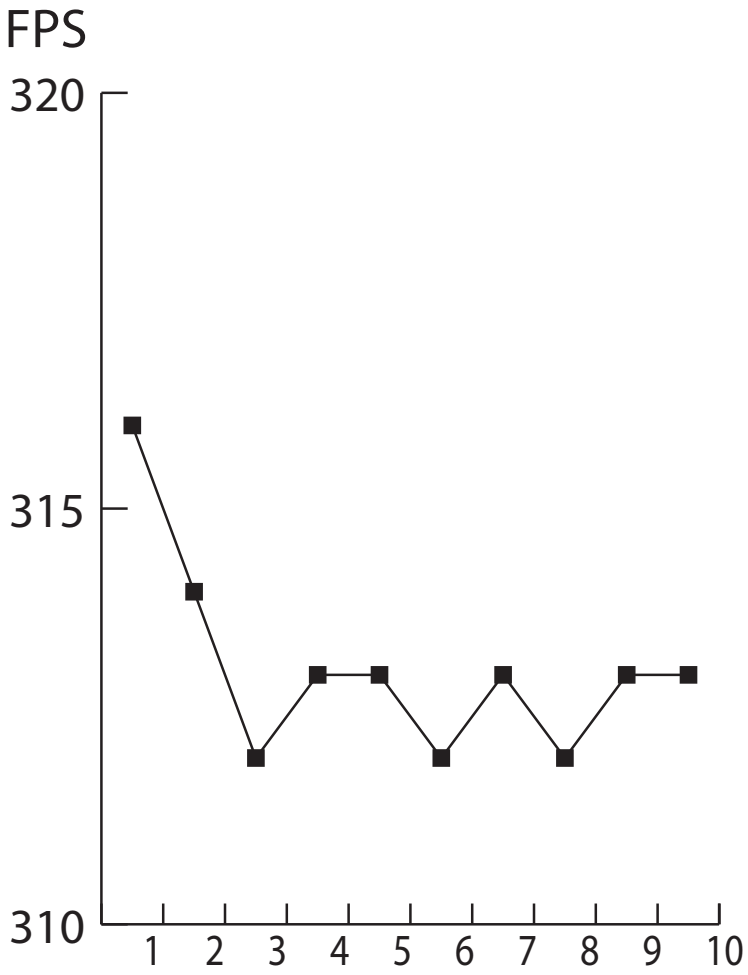


Figure 6.2: FPS graph of sketch treatment and agent traversal

agent is also traversing the original path. From 0 - 5 seconds, the frame rate fluctuates between 304 - 305 FPS. At 5 seconds the frame-rate reaches 305 FPS. The intersection tests and the splicing of the paths begin. The lowest FPS recorded is 304. After the splicing, the agent continues traversing the path and as no input is being read from the device, the frame rate increases to 306 FPS. Of all the commands possible with the sketch system, editing is the worst with regard to performance. Regardless, the frame rate is high enough that there is no perceivable drop in frame rate in the application itself.

Next to be benchmarked is the group cooperative pathfinding. A sketch stroke is drawn and each agent thereafter must add a path to the reservation table and begin traversing it. Figure 6.4 shows

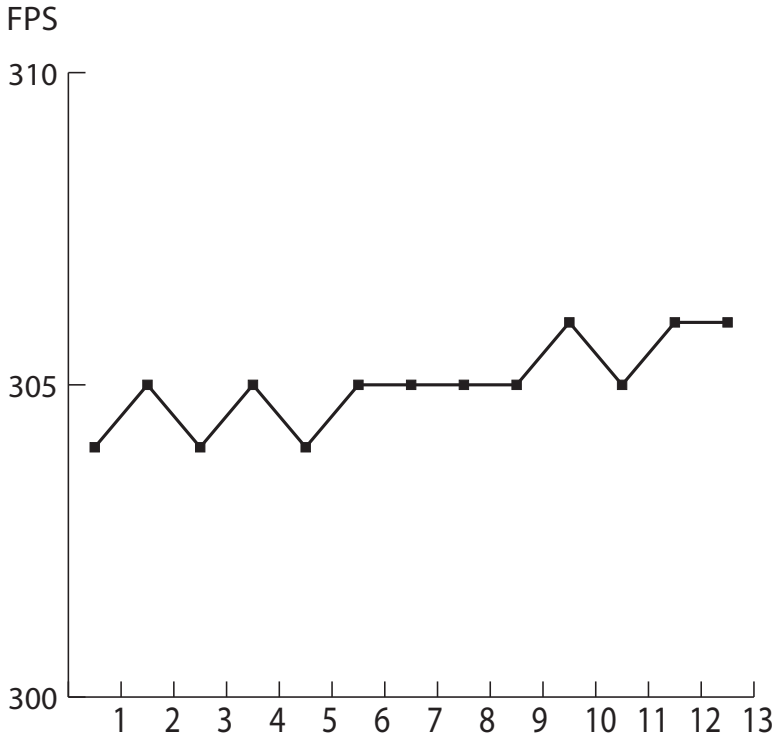
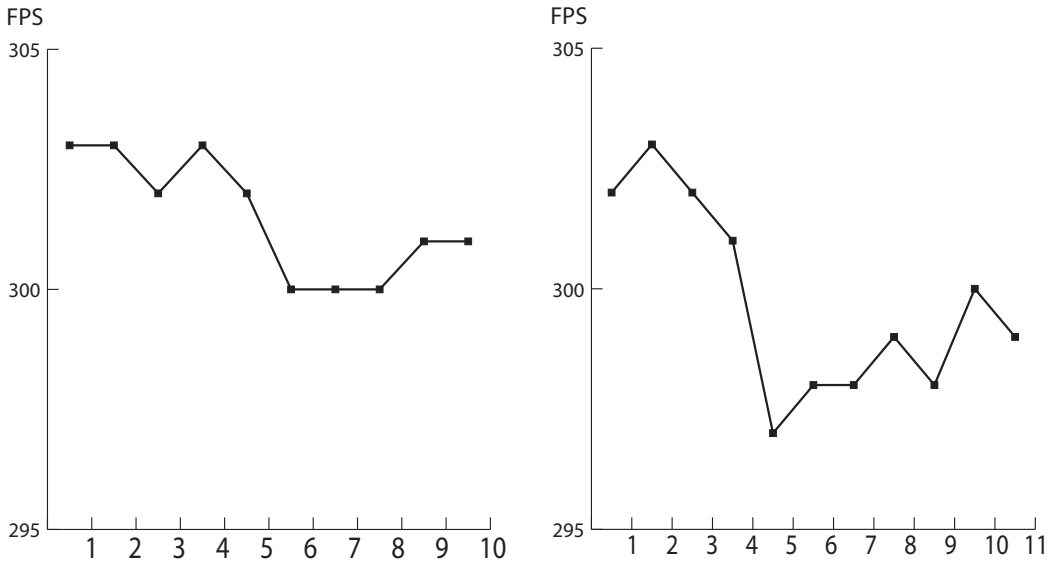


Figure 6.3: FPS graph of sketch-based editing with agent traversal

two graphs, both displaying the FPS for cooperative pathfinding with different numbers of agents. Figure 6.4a shows the FPS for three agents. The minimum frame rate is 300 as the reservation tables are being created and A* is being run. In figure 6.4b, it can seen that the frame rate also drops to a minimum of 297 but the time the drop occurs over is longer with the extra agents causing longer computation time. A* needs to be run for two extra agents and their paths need to be added to the reservation table. Once all paths have been created and reserved the frame rate rises as the agents traverse their paths. As 6.4a only has three agents, the frame rate is higher at 301 FPS while the other with five agents rises to 300 FPS.

6.2 Overview

As the results show, the inclusion of sketch-based interaction with pathfinding is real-time, with the frame rates being substantially higher than 30 FPS which most interactive entertainment applications such as games would aim for. The addition of the ray acceleration structure helped improve the frame rate immensely. Without its addition, frame rates were reaching as low as 245 FPS for the more



(a) 3 agents (b) 5 agents

Figure 6.4: FPS graph of Cooperative Pathfinding

intensive tasks, while with it the lowest frame rate recorded was 297 FPS.

7
Conclusion

This chapter outlines the conclusion to the project and mentions possible future work that could be made to the system, should it be continued in the future.

7.1 Conclusions

This project had a number of aims when it was started. It was designed to incorporate the use of a sketch-based input device with pathfinding and to do so in an interactive real-time application. It allows the use of sketch strokes drawn by the user to create traversable paths for agents over varying terrain types. A treatment process was created for sketch strokes to allow their use by agents as paths. A gesture recognition system was built which made it possible to use a stylus and sketch strokes for modifying paths and to allow for multiple selection of agents, both using varying gestures drawn by a user. For group movement, it extended agent awareness, so their paths could be planned relative to each other, all under the direction of sketch strokes. The project gives the user the possibility of customising the systems pathfinder by limiting parts of the map it could plan for through sketch.

Overall the system shows that sketch-based input is a viable way for controlling agent movement and it does this in real-time, allowing for its implementation in interactive applications such as games. It has achieved the goals set out for this project and its integration into interactive applications will lead to greater user control of agent movement than is currently available today.

7.2 Future Work

This section contains information on possible developments that could be added to the project should it continue. Some are entirely new additions to the project itself while others are optimisations which would improve the current project as it is now.

7.2.1 Multiple Point Pathfinding

Ways of using sketch-input with paths could be expanded. For instance, rather than designate a particular position for an agent to go to, you could designate an area where any position in the area could be the target. A self-enclosed sketch could be drawn designating this area as the target and the systems pathfinder would have to find a path to somewhere in this area. As the shortest path should always be chosen, the system would have to find the shortest path to multiple points and compare them. A* can only perform one target point at a time but algorithms like the Floyd-Warshall algorithm do not. Floyd-Warshall could find paths to multiple targets and compile the shortest path[47]. It is also parallisable so it could be accelerated on the GPU[17]. If the environment in which the algorithm was running changed, a path which was the shortest may not be so any more and another path to another position in the area defined could be used.

7.2.2 Dynamic Environment

The pathfinding algorithms could be expanded to take into account dynamic environments. This could have interesting implications for user defined paths as they could possibly be blocked by an obstacle after the user has drawn them. The system would have to deal with this obstacle by possibly informing the user that their path, although viable when it was drawn is no longer so.

A form of D* could be implemented where the cost of nodes are constantly changing due to the dynamic environment. This would allow paths to dynamically redrawn taking to account the environment. A negative is that D* like solutions can have very high memory requirements and might not be suitable for real-time applications which are already conscious about performance like the game industry.

7.2.3 Sweep Line Algorithm

In this project the gesture recognition for path commands is built around line segment intersection tests. The number of line segments are quite low during run time as they are limited by the size

of the paths themselves and checks for intersections do not occur often as they are only checked for intersections when the user draws a sketch stroke. Since the intersections are relatively infrequent, the intersection tests were not optimised but when there are 100's possibly thousands of 1000's selected on screen, such intersection tests could be extremely inefficient.

An optimisation which could be made would be the sweep line algorithm. Described by Shamos and Hoey[39] in 1976, it describes an algorithm for efficiently checking for line intersections. Given n line segments, the sweep line algorithm completes the same checks in O(n log n) time. It works by sweeping over the range of values eliminating some and not others for intersection tests as it moves. As the points in this project are 2D, if a line moves from the start of the plane to the right, it checks to see which segments it is currently intersecting as it moves then it checks all of these against each other. When it sweeps over a new segment it checks this against the others it is currently sweeping over. When it has passed over a line segment, it will no longer check for intersections against it. In this way the sweep line algorithm narrows down the segments it runs intersection tests on making it faster at running them.

7.2.4 Pathfinding

For A* itself, there is some possible optimisations which could be used. A binary heap data structure would improve performance. As a heuristic, the agents orientation could be taken into account leading to some nicer turns for agents themselves.

A
Sketch Camera Controls

This section describes how the sketch stylus controls the camera. Figure A.1 shows a diagram with the positions needed and if there is buttons to press for activating certain movements.

The instructions for using the controls are - To pan, the cursor from the stylus needs to be placed into one of the outer boundary areas at the edge of the screen. For rotating, the first button on the stylus needs to be pressed with the cursor being placed on the left and right of the screen to rotate left and right respectively. To zoom, the second button needs to be pressed with the cursor being placed on the top or bottom of the screen to zoom in and out respectively.

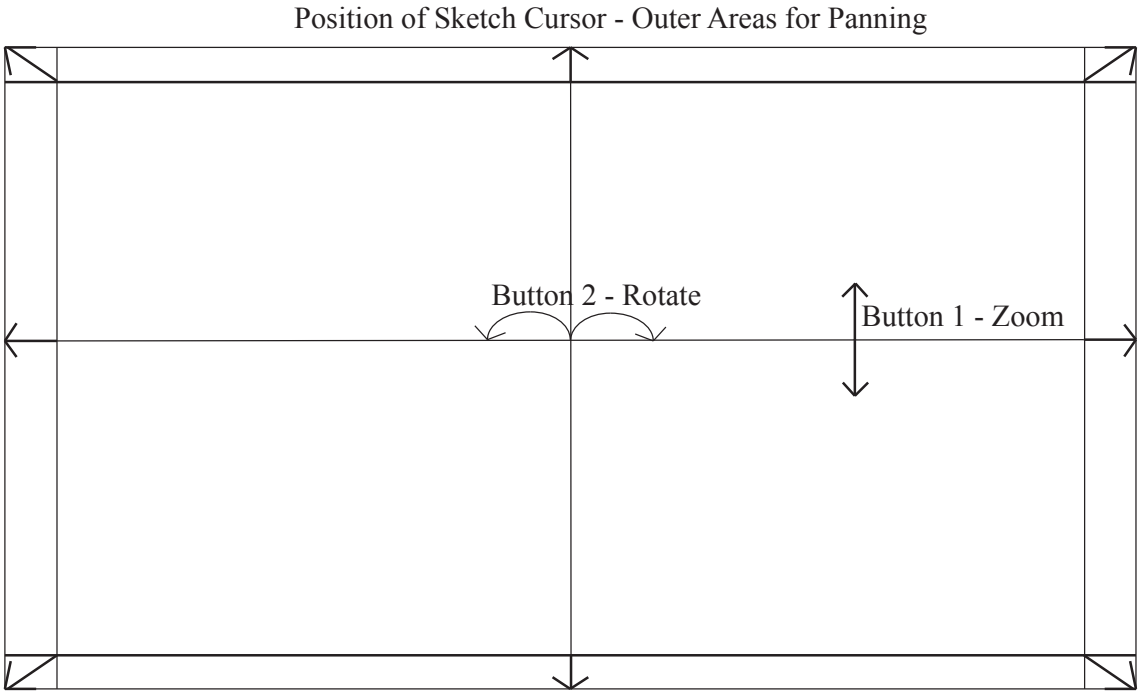


Figure A.1: Stylus based camera controls

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Christine Alvarado, Randall Davis, and All Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1365–1371. Morgan Kaufmann Publishers, 2001.

[3] Fabricio Anastacio, Przemyslaw Prusinkiewicz, and Mario Costa Sousa. Sketch-based interfaces and modeling (sbim): Sketch-based parameterization of l-systems using illustration-inspired construction lines and depth modulation. *Computers and Graphics*, 33(4):440–451, 2009.

[4] Danilo Avola, Andrea Buono, Giorgio Gianforme, and Stefano Paolozzi. A novel recognition approach for sketch-based interfaces. In *ICIAP '09: Proceedings of the 15th International Conference on Image Analysis and Processing*, pages 1015–1024, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Danilo Avola, Maria Chiara Caschera, Fernando Ferri, and Patrizia Grifoni. Ambiguities in sketch-based interfaces. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 290b, Washington, DC, USA, 2007. IEEE Computer Society.

[6] William Baxter. bbtablet. `http://www.billbaxter.com/projects/bbtablet/index.html`, September 2010.

[7] Beepa. FRAPS. `http://www.fraps.com/`, September 2010.

[8] Alexander Blessing, T. Metin Sezgin, Relja Arandjelovic, and Peter Robinson. A multimodal interface for road design. Technical report, University of Cambridge, 2009.

[9] GLEST Community. Glest. `http://glest.org/en/index.php`, September 2010.

[10] Matthew T. Cook and Arvin Agah. A survey of sketch-based 3-d modeling techniques. Technical report, University of Kansas, 2009.

[11] Mathias Eitz, Olga Sorkine, and Marc Alexa. Sketch based image deformation. In *Proceedings of Vision, Modeling and Visualization (VMV)*, pages 135–142, 2007.

[12] Eurographics. Preface. In *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, June 2004.

[13] Eurographics. Article count. In *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, June 2008.

[14] S. D. Goodwin, S. Menon, and R. G. Price. Pathfinding in open terrain, 2006.

[15] The Logic Group. Wintab. `http://www.logicgroup.com/WintabDriver.htm`, September 2010.

[16] Tracy Hammond and Randall Davis. Ladder, a sketching language for user interface developers. *Computers and Graphics*, 29(4):518–532, 2005.

[17] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA.

[18] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, July 1968.

[19] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3d freeform design, 1999.

[20] Levent Burak Kara, Chris M. D'Eramo, and Kenji Shimada. Pen-based styling design of 3d geometry using concept sketches and template models. Technical report, Carnegie Mellon University, 2006.

[21] Sven Koenig and Maxim Likhachev. D*lite. In *Eighteenth national conference on Artificial intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[22] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artif. Intell.*, 155(1-2):93–146, 2004.

[23] Natallia Kokash. An introduction to heuristic algorithms. Technical report, Department of Informatics and Telecommunications. University of Trento, Italy, 2005.

[24] Jeehyung Lee and Thomas Funkhouser. Sketch-based search and composition of 3d models. In *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, 2008.

[25] WeeSan Lee, Levent Burak Kara, and Thomas F. Stahovich. An efficient graph-based recognizer for hand-drawn symbols. *Computers and Graphics*, 31(4):554–567, 2007.

[26] Hod Lipson and Moshe Shpitalni. Correlation-based reconstruction of a 3d object from a single freehand sketch. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 44, New York, NY, USA, 2007. ACM.

[27] Ellen Yi luen Do and Mark D. Gross. Drawing as a means to design reasoning. *AI and Design*, 1996.

[28] Tiago Lemos de Araujo Machado, Alex Sandro Gomes, and Marcelo Walter. A comparison study: Sketch-based interfaces versus wimp interfaces in three dimensional modeling tasks. In *LA-WEB '09: Proceedings of the 2009 Latin American Web Congress (la-web 2009)*, pages 29–35, Washington, DC, USA, 2009. IEEE Computer Society.

[29] C Mao, S.F. Qin, and D.K. Wright. A sketch-based gesture interface for rough 3d stick figure animation. Technical report, Brunel University, 2005.

[30] James McCrae and Karan Singh. Sketch-based path design. In *GI '09: Proceedings of Graphics Interface 2009*, pages 95–102, Toronto, Ont., Canada, Canada, 2009. Canadian Information Processing Society.

[31] Microsoft. Kinect. `www.xbox.com/kinect`, September 2010.

[32] Ian Millington. *Artificial Intelligence For Games*. Morgan Kaufmann, 2006.

[33] G. Nataneli and P. Faloutsos. Sketch-based facial animation. Technical report, University of California, Los Angeles, 2006.

[34] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Fibermesh: designing freeform surfaces with 3d curves. *ACM Trans. Graph.*, 26(3):41, 2007.

[35] NINTENDO. Wii remote. `http://www.nintendo.com/wii`, September 2010.

[36] L. Olsen, F.F. Samavati, M.C. Sousa, and J. Jorge. A taxonomy of modeling techniques using sketch-based interfaces. In *Eurographics 2008 State-of-the-Art Report (EG'08 STAR)*, 2008.

[37] Luke Olsen, Faramarz F. Samavati, Mario C. Sousa, and Joaquim A. Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33(1):85–103, February 2009.

[38] Dean Rubine. Specifying gestures by example. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 329–337, New York, NY, USA, 1991. ACM.

[39] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. pages 208 –215, oct. 1976.

[40] Hyojong Shin and Takeo Igarashi. Iagarashi t.: Magic canvas: Interactive design of a 3-d scene prototype from freehand sketches. In *In Proceedings of Graphics Interface*, 2007.

[41] David Silver. Cooperative pathfinding. In S. Rabin, editor, *Game AI Programming Wisdom 3*, Cambridge, MA, 2006. Charles River.

[42] IEEE Computer Society. Stroke-input methods for immersive styling environments. In *SMI '04: Proceedings of the Shape Modeling International 2004*, pages 275–283, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Anthony Stentz and Is Carnegle Mellon. Optimal and efficient path planning for unknown and dynamic environments. *International Journal of Robotics and Automation*, 10:89–100, 1993.

[44] Xiaoxun Sun, William Yeoh, and Sven Koenig. Dynamic fringe-saving A*. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 891–898, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[45] Paul Tozour. AI Game Programming Wisdom 2. Charles River Media, 2004.

[46] Wacom. Bamboo pen tablet. `http://www.wacom.com/bamboo/`, September 2010.

[47] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[48] Ji wung Choi, Renwick Curry, and Gabriel Elkaim. Path planning based on bezier curve for autonomous ground vehicles. *World Congress on Engineering and Computer Science, Advances in Electrical and Electronics Engineering - IAENG Special Edition of the*, 0:158–166, 2008.

[49] Chen Yang, Dana Sharon, and Michiel van de Panne. Sketch-based modeling of parameterized objects. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 89, New York, NY, USA, 2005. ACM.

[50] Kwangjin Yang and S. Sukkarieh. An analytical continuous-curvature path-smoothing algorithm. *Robotics, IEEE Transactions on*, 26(3):561 –568, jun. 2010.

[51] Shane W. Zamora and Eyrn A. Eyjlfsdttir. Circuitboard: Sketch-based circuit design and analysis, 2009.